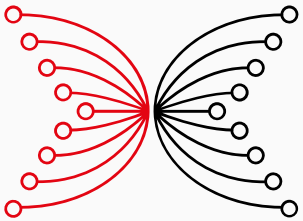
Type system

Haskell and Cryptocurrencies

Dr. Andres Löh, Well-Typed LLP
Dr. Lars Brünjes, IOHK
Dr. Polina Vinogradova, IOHK

2019-02-19



INPUT | OUTPUT

Goals

- Type checking vs. type inference
- Damas-Hindley-Milner type system
- Implementation of a type checker
- A word on extensions

Type checking vs. type inference

Typing an expression

```
map toUpper input
```

Typing an expression

```
map toUpper input
```

In order to say anything, we should know the types of the free variables:

```
map     :: (a -> b) -> [a] -> [b]
toUpper :: Char -> Char
input   :: [Char]
```

Typing an expression

```
map toUpper input
```

In order to say anything, we should know the types of the free variables:

```
map     :: (a -> b) -> [a] -> [b]
toUpper :: Char -> Char
input   :: [Char]
```

A finite map associating identifiers with types is called a type environment.

Type checking

Given an expression and a type environment for its free variables and a type, decide whether the expression has the given type.

Type checking

Given an expression and a type environment for its free variables and a type, decide whether the expression has the given type.

Example:

```
map toUpper input
```

With the environment from just before and type `[Char]`, we can check that the expression has this type.

Type inference

Given an expression and a type environment for its free variables, find a valid (better: the most general) type of the expression.

Type inference

Given an expression and a type environment for its free variables, find a valid (better: the most general) type of the expression.

Example:

```
map toUpper input
```

With the environment from just before, we can infer the type of the expression to be `[Char]`.

Checking vs. inference

For a simple expression involving just function applications, both checking and inference are quite simple.

Checking vs. inference

For a simple expression involving just function applications, both checking and inference are quite simple.

Things get more interesting once we add abstraction.

Abstraction example

```
\ x -> x ++ x
```

Type environment:

```
(++) :: [a] -> [a] -> [a]
```

Abstraction example

```
\ x -> x ++ x
```

Type environment:

```
(++) :: [a] -> [a] -> [a]
```

Checking this to be of type `[Char] -> [Char]` is easy: We can immediately see that `x :: [Char]` and verify that `(++)` is used correctly and the result type matches.

Abstraction example

```
\ x -> x ++ x
```

Type environment:

```
(++) :: [a] -> [a] -> [a]
```

Checking this to be of type `[Char] -> [Char]` is easy: We can immediately see that `x :: [Char]` and verify that `(++)` is used correctly and the result type matches.

Inferring the type requires us to make an assumption for the type of `x`. The assumption determines whether inference succeeds or not.

Assumptions and constraints

```
\ x -> x ++ x
```

Assumptions and constraints

```
\ x -> x ++ x
```

If we assume `x :: Int`, inference fails, because `(++)` requires a list.

Assumptions and constraints

```
\ x -> x ++ x
```

If we assume `x :: Int`, inference fails, because `(++)` requires a list.

If we assume `x :: [Char]` or `x :: [Int]`, inference succeeds, but with different result types.

Assumptions and constraints

```
\ x -> x ++ x
```

If we assume `x :: Int`, inference fails, because `(++)` requires a list.

If we assume `x :: [Char]` or `x :: [Int]`, inference succeeds, but with different result types.

If we assume `x :: [a]`, inference succeeds with (intuitively) the best possible type `[a] -> [a]`.

Key idea of inference

```
\ x -> x ++ x
```

Key idea of inference

```
\ x -> x ++ x
```

We assume `x :: v1` for some inference variable `v1`.

Key idea of inference

```
\ x -> x ++ x
```

We assume `x :: v1` for some inference variable `v1`.

Use of `x` constrains `v1`. Using `x` as an argument to `(++)` means that

```
v1 = [v2]
```

must hold for some new inference variable `v2`.

Key idea of inference

```
\ x -> x ++ x
```

We assume `x :: v1` for some inference variable `v1`.

Use of `x` constrains `v1`. Using `x` as an argument to `(++)` means that

```
v1 = [v2]
```

must hold for some new inference variable `v2`.

Therefore, the type of the expression is `[v2] -> [v2]`.

Key idea of inference

```
\ x -> x ++ x
```

We assume `x :: v1` for some inference variable `v1`.

Use of `x` constrains `v1`. Using `x` as an argument to `(++)` means that

```
v1 = [v2]
```

must hold for some new inference variable `v2`.

Therefore, the type of the expression is `[v2] -> [v2]`.

Inference variables that are unconstrained at the end can be generalized to normal type variables, so the type is `[a] -> [a]`.

Difficulty of inference

- Making assumptions in such a way that they do not disallow potentially valid solutions is, in general, difficult.

Difficulty of inference

- Making assumptions in such a way that they do not disallow potentially valid solutions is, in general, difficult.
- On the other hand, checking remains easy as long as we have explicit types for all abstractions.

Difficulty of inference

- Making assumptions in such a way that they do not disallow potentially valid solutions is, in general, difficult.
- On the other hand, checking remains easy as long as we have explicit types for all abstractions.
- Languages with very powerful type systems can still be checked as long as they are explicitly typed.

Difficulty of inference

- Making assumptions in such a way that they do not disallow potentially valid solutions is, in general, difficult.
- On the other hand, checking remains easy as long as we have explicit types for all abstractions.
- Languages with very powerful type systems can still be checked as long as they are explicitly typed.
- Inference is still surprisingly powerful, but has its limits.

Damas-Hindley-Milner (DHM) type system

Features of DHM

- Type inference for a lambda calculus with abstraction, application and let-bindings.
- Types can be polymorphic.
- Most general types will be inferred.
- Inference works in the most common situations, with only a few exceptions (such as polymorphic recursion).

What we will do

- Formal description of DHM.
- Implementation of a DHM for a simple language in Haskell.
- Discuss how DHM can be extended from the simple lambda calculus with let bindings to “full” Haskell.

The language of types

Monotypes:

$$
\begin{array}{lll}
\tau ::= \alpha & & \text{(type variable)} \\
\quad \mid \quad T_n \ \tau_1 \ \ldots \ \tau_n & & \text{(constructor application)} \\
\quad \mid \quad \tau_1 \rightarrow \tau_2 & & \text{(function type)}
\end{array}
$$

The language of types

Monotypes:

$$
\begin{array}{lll}
\tau ::= \alpha & & \text{(type variable)} \\
\quad \mid \quad T_n\ \tau_1 \ldots \tau_n & & \text{(constructor application)} \\
\quad \mid \quad \tau_1 \to \tau_2 & & \text{(function type)}
\end{array}
$$

Polytypes (also known as type schemes):

$$\sigma ::= \forall \alpha_1 \ldots \alpha_n.\ \tau$$

The language of types

Monotypes:

$$
\begin{array}{lll}
\tau ::= \alpha & & \text{(type variable)} \\
\quad \mid \quad T_n\ \tau_1 \ldots \tau_n & & \text{(constructor application)} \\
\quad \mid \quad \tau_1 \to \tau_2 & & \text{(function type)}
\end{array}
$$

Polytypes (also known as type schemes):

$$\sigma ::= \forall \alpha_1 \ldots \alpha_n.\ \tau$$

Monotypes are polytypes

Note that in the definition of *polytypes*, $n = 0$ is allowed, i.e. each *monotype* is a *polytype*.

Examples: constructor application

Datatypes are all covered by the *constructor application* case:

$$T_n\ \tau_1\ \ldots\ \tau_n$$

Examples: constructor application

Datatypes are all covered by the *constructor application* case:

$$T_n\ \tau_1\ \ldots\ \tau_n$$

For example,

```
Int
Either α β
[α]
```

are constructor applications:

- `Int` is a constructor of arity `0`.
- `Either` is a constructor of arity `2`.
- `[]` is a constructor of arity `1`.

Admissible datatypes

The admissible type constructors and their arities are assumed to be fixed and given in advance, unlike in Haskell, where they can be defined using `data`.

Examples: polymorphic types

Haskell polymorphic types are all *type schemes* with the usually implicit universal quantifiers being made explicit:

$$\forall \alpha \beta.\ (\alpha \to \beta) \to [\alpha] \to [\beta]$$

corresponds to

```
(a -> b) -> [a] -> [b]
```

Examples: polymorphic types

Haskell polymorphic types are all *type schemes* with the usually implicit universal quantifiers being made explicit:

$$\forall \alpha \beta.\ (\alpha \to \beta) \to [\alpha] \to [\beta]$$

corresponds to

```
(a -> b) -> [a] -> [b]
```

Note: *type variables* are always quantified on the outside.

Type environments

$$
\begin{array}{lll}
\Gamma ::= & \epsilon & \text{(empty environment)} \\
\mid & \Gamma, x :: \sigma & \text{(extended environment)}
\end{array}
$$

Type environments

$$
\begin{array}{lll}
\Gamma ::= & \epsilon & \text{(empty environment)} \\
\mid & \Gamma, x :: \sigma & \text{(extended environment)}
\end{array}
$$

Environments are modelled as lists associating identifiers with type schemes. Later bindings shadow earlier bindings.

Type environments

$$
\begin{array}{lll}
\Gamma ::= \epsilon & & \text{(empty environment)} \\
\quad \mid & \Gamma, x :: \sigma & \text{(extended environment)}
\end{array}
$$

Environments are modelled as lists associating identifiers with type schemes. Later bindings shadow earlier bindings.

In the theoretical literature, `:` is being used to denote the "has-type" relation. We stick to the Haskell `::` notation, to avoid unnecessary confusion.

The initial environment

We can make arbitrary assumptions in our initial environment and thereby model everything that is currently in scope:

```
  ϵ
, True   :: Bool
, False  :: Bool
, filter :: (α → Bool) → [α] -> [α]
, (++)   :: [α] → [α] → [α]
, []     :: [α]
, (:)    :: α → [α] → [α]
, ...
```

The language of terms

$$
\begin{array}{lll}
e ::= x & & \text{(variables)} \\
\quad \mid & (e_1\ e_2) & \text{(application)} \\
\quad \mid & \lambda x \to e & \text{(lambda abstraction)} \\
\quad \mid & \textbf{let}\ x = e_1\ \textbf{in}\ e_2 & \text{(non-recursive let)}
\end{array}
$$

The language of terms

$$
\begin{array}{lll}
e ::= x & & \text{(variables)} \\
\quad | \quad (e_1\ e_2) & & \text{(application)} \\
\quad | \quad \lambda x \to e & & \text{(lambda abstraction)} \\
\quad | \quad \textbf{let}\ x = e_1\ \textbf{in}\ e_2 & & \text{(non-recursive let)}
\end{array}
$$

Let

Remember that in the *untyped* lambda calculus,

$\textbf{let}\ x = e_1\ \textbf{in}\ e_2$ was just syntactic sugar for

$(\lambda x \to e_2)\ e_1$ – we will see that in DHM, this is different

Recursion

General recursion is possible by simply adding

$$\texttt{fix} :: \forall\alpha.(\alpha \to \alpha) \to \alpha$$

to the intial environment.

Modelling (simple) pattern matching

```
caseList :: [α] → β → (α → [α] → β) → β
```

Modelling (simple) pattern matching

```
caseList :: [α] → β → (α → [α] → β) → β
```

```
let
  mapf = λrfx →
    caseList x
      []
      (λyz → f y : r f z)
  map = fix mapf
in
  ...
```

Type judgement

$$\Gamma \vdash e :: \tau$$

The $\vdash$ and $::$ here are just syntax. This is a *relation* between a type environment, an expression, and a type, and reads

"Under environment Γ, expression e has type τ."

Type judgement

$$\Gamma \vdash e :: \tau$$

The $\vdash$ and $::$ here are just syntax. This is a *relation* between a type environment, an expression, and a type, and reads

"Under environment Γ, expression e has type τ."

One of the peculiarities of DHM is: the type environment maps identifiers to *polytypes*, but the judgements are for *monotypes*.

The type system

Type system vs. algorithm

Type systems are often described in several different ways, with different goals:

- a specification, optimised for clarity (and type checking),
- an algorithm, optimised for ease of implementation (and type inference).

Type system vs. algorithm

Type systems are often described in several different ways, with different goals:

- a specification, optimised for clarity (and type checking),
- an algorithm, optimised for ease of implementation (and type inference).

We'll present a specification of DHM first, and an actual algorithm for type inference afterwards.

Variables

$$\frac{x :: \sigma \in \Gamma \qquad \tau = \textbf{instantiate}(\sigma)}{\Gamma \vdash x :: \tau}$$

This is an inference rule: if the prerequisites (above the bar) hold, then the conclusion (below the bar) holds as well.

Variables

$$\frac{x :: \sigma \in \Gamma \qquad \tau = \textbf{instantiate}(\sigma)}{\Gamma \vdash x :: \tau}$$

This is an inference rule: if the prerequisites (above the bar) hold, then the conclusion (below the bar) holds as well.

The helper function **instantiate** substitutes the quantified variables with monotypes in the body of the type scheme.

Example instantiation of the rule for variables

Assume $\Gamma = \epsilon, (:) :: \forall\alpha.\ (\alpha \to [\alpha] \to [\alpha]), \texttt{one} :: \texttt{Int}$.

$$\frac{(:) \in \Gamma \qquad \texttt{Int} \to [\texttt{Int}] \to [\texttt{Int}] = \mathsf{instantiate}(\alpha \to [\alpha] \to [\alpha])}{\Gamma \vdash (:) :: \texttt{Int} \to [\texttt{Int}] \to [\texttt{Int}]}$$

Application

$$\frac{\Gamma \vdash e_1 :: \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 :: \tau_1}{\Gamma \vdash (e_1\ e_2) :: \tau_2}$$

Example instantiation of the rule for application

Assume $\Gamma = \epsilon, (:) :: \forall\alpha.\ (\alpha \to [\alpha] \to [\alpha]), \texttt{one} :: \texttt{Int}$.

$$\frac{\Gamma \vdash (:) :: \texttt{Int} \to [\texttt{Int}] \to [\texttt{Int}] \qquad \Gamma \vdash \texttt{one} :: \texttt{Int}}{\Gamma \vdash ((:)\ \texttt{one}) :: [\texttt{Int}] \to [\texttt{Int}]}$$

Full derivation example

Assume $\Gamma = \epsilon, \texttt{(+)} :: \texttt{Int -> Int -> Int}, \texttt{1} :: \texttt{Int}$.

$$\dfrac{\dfrac{\dfrac{\texttt{(+)} :: \texttt{Int} \to \texttt{Int} \to \texttt{Int} \in \Gamma}{\Gamma \vdash \texttt{(+)} :: \texttt{Int} \to \texttt{Int} \to \texttt{Int}} \quad \dfrac{\texttt{1} :: \texttt{Int} \in \Gamma}{\Gamma \vdash \texttt{1} :: \texttt{Int}}}{\Gamma \vdash \texttt{(+) 1} :: \texttt{Int} \to \texttt{Int}} \quad \dfrac{\texttt{1} :: \texttt{Int} \in \Gamma}{\Gamma \vdash \texttt{1} :: \texttt{Int}}}{\Gamma \vdash \texttt{1 + 1} :: \texttt{Int}}$$

Abstraction

$$\frac{\Gamma, x :: \tau_1 \vdash e :: \tau_2}{\Gamma \vdash \lambda x \to e :: \tau_1 \to \tau_2}$$

Example instantiation of the rule for abstraction

Assume $\Gamma = \epsilon, (+) :: \texttt{Int} \to \texttt{Int} \to \texttt{Int}$.

$$\frac{\Gamma, x :: \texttt{Int} \vdash x + x :: \texttt{Int}}{\Gamma \vdash \lambda x \to x + x :: \texttt{Int} \to \texttt{Int}}$$

Let

$$\frac{\Gamma \vdash e_1 :: \tau_1 \qquad \sigma = \textsf{generalise}(\Gamma, \tau_1) \qquad \Gamma, x :: \sigma \vdash e_2 :: \tau_2}{\Gamma \vdash \textsf{let } x = e_1 \textsf{ in } e_2 :: \tau_2}$$

The helper function **generalise** quantifies a monotype over all free type variables that do not occur free in the environment.

Example instantiation of the rule for let

Assume $\Gamma = \epsilon, \texttt{0} :: \texttt{Int}$.

$$\frac{\begin{array}{c}\Gamma \vdash \lambda x \to x :: \mathsf{a} \to \mathsf{a} \\ \forall \mathsf{a}.\ \mathsf{a} \to \mathsf{a} = \mathsf{generalise}(\Gamma, \mathsf{a} \to \mathsf{a}) \\ \Gamma, f :: \forall \mathsf{a}.\ (\mathsf{a} \to \mathsf{a}) \vdash (f\ f\ \texttt{0}) :: \texttt{Int}\end{array}}{\Gamma \vdash \mathsf{let}\ f = \lambda x \to x\ \mathsf{in}\ f\ f\ \texttt{0} :: \texttt{Int}}$$

Why restrict generalisation with the environment?

Consider:

$$\lambda x \to \textsf{let } y = x : [\,] \textsf{ in } 0 : y$$

Why restrict generalisation with the environment?

Consider:

$\lambda x \to \textbf{let}\ y = x : [\,]\ \textbf{in}\ 0 : y$

If we did *not* restrict generalisation with the environment, we could prove that that expression has type `a → [Int]`, where it clearly should have type `Int → [Int]`.

Why restrict generalisation with the environment?

Consider:

$$\lambda x \to \mathsf{let}\ y = x : [\,]\ \mathsf{in}\ 0 : y$$

If we did *not* restrict generalisation with the environment, we could prove that that expression has type `a → [Int]`, where it clearly should have type `Int → [Int]`.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dots}{\Gamma, x :: a \vdash x : [\,] :: [a]} \qquad \dfrac{\dots}{\Gamma, x :: a, y :: \boxed{\forall a.[a]} \vdash 0 : y :: [\mathtt{Int}]}}{\Gamma, x :: a \vdash \mathsf{let}\ y = x : [\,]\ \mathsf{in}\ 0 : y :: [\mathtt{Int}]}}{\Gamma \vdash \lambda x \to \mathsf{let}\ y = x : [\,]\ \mathsf{in}\ 0 : y :: a \to [\mathtt{Int}]}}$$

The power of Let

- In the *Untyped Lambda Calculus*, $\textbf{let}\ x = e\ \textbf{in}\ f$ was merely syntactic sugar for $(\lambda x \to f)\ e$.
- In DHM, Let adds strictly more expressiveness.
- For an example, assume $0 :: \mathit{Int} \in \Gamma$, and consider $\textbf{let}\ f = \lambda x \to x\ \textbf{in}\ (f\ f)\ \texttt{0}$, which type-checks and has type $\texttt{Int}$.
- The "desugared" term $(\lambda f \to (f\ f)\ \texttt{0})\ (\lambda x \to x)$, however, does not.
- The crucial point here is that f is instantiated at two different monotypes in the body, first at $(\texttt{Int} \to \texttt{Int}) \to (\texttt{Int} \to \texttt{Int})$, then at $\texttt{Int} \to \texttt{Int}$.

Reduction

- For the *Untyped Lambda Calculus*, we defined how to *reduce* a lambda expression.
- The same reduction rules apply to DHM, but we have to explain how to reduce a (sub-)expression using Let.
- $\textbf{let}\ x = e\ \textbf{in}\ f$ is a redux and reduces to $f\ [x := e]$.
- The relation between type-checking and reduction is as follows: If an expression e is well-typed and reduces to an expression f, then f is also well-typed and has the same type as e.

Algorithm W

Algorithm W

The specification still leaves many things open with respect to type inference:

- What types do we instantiate quantified variables to when instantiating?
- What type do we assume for the variable in the abstraction?
- How do we achieve that the type of the function and the type of the argument line up properly in an application?

Algorithm W

The specification still leaves many things open with respect to type inference:

- What types do we instantiate quantified variables to when instantiating?
- What type do we assume for the variable in the abstraction?
- How do we achieve that the type of the function and the type of the argument line up properly in an application?

We now present an algorithm (called Algorithm W in the original paper by Damas and Milner) compatible with the specification that can easily be implemented and actually infers a type for a given term.

Substitutions

During the algorithm, we build up a substitution θ, which maps type variables to monotypes.

Substitutions

During the algorithm, we build up a substitution θ, which maps type variables to monotypes.

Building substitutions:

```
id
```

The identity substitution.

$$[\alpha \mapsto \tau]$$

The substitution that maps α to τ and leaves all other variables unchanged.

Applying substitutions

$\theta\tau$

Applies the substitution θ to the (free variables in) type τ.

$\theta\Gamma$

Applies the substitution θ to all types in Γ.

Composing substitutions

$\theta_2 \circ \theta_1$

The substitution θ_1 followed by the substitution θ_2.

Variables

The algorithm W maps an environment and a term to a substitution and a monotype.

$$\mathsf{W}(\Gamma, x) = (\mathsf{id}, \mathsf{instantiate}(\sigma))$$
$$\text{where}$$
$$x :: \sigma \in \Gamma$$

Variables

The algorithm W maps an environment and a term to a substitution and a monotype.

$$\mathsf{W}(\Gamma, x) = (\mathsf{id}, \mathsf{instantiate}(\sigma))$$
$$\text{where}$$
$$x :: \sigma \in \Gamma$$

The function $\mathsf{instantiate}$ is as before, only that we replace all quantified types with fresh type variables.

Abstraction

$$\mathsf{W}(\Gamma, \lambda x \to e) = (\theta, \theta(\alpha \to \tau))$$

where

$$(\theta, \tau) = \mathsf{W}((\Gamma, x :: \alpha), e)$$

α fresh

For a lambda abstraction, we assume a fresh type variable as the type of the variable.

Application

$$
\begin{aligned}
&\mathsf{W}(\Gamma, (e_1\ e_2)) = (\theta_3 \circ \theta_2 \circ \theta_1, \theta_3\alpha) \\
&\quad \text{where} \\
&\qquad (\theta_1, \tau_1) = \mathsf{W}(\Gamma, e_1) \\
&\qquad (\theta_2, \tau_2) = \mathsf{W}(\theta_1\Gamma, e_2) \\
&\qquad \theta_3 = \mathbf{unify}(\theta_2\tau_1, \tau_2 \to \alpha) \\
&\qquad \alpha \text{ fresh}
\end{aligned}
$$

The function **unify** tries to find a substitution that makes the two given types equal. We will look at **unify** after the case for let bindings.

Let

$$
\begin{aligned}
&\mathsf{W}(\Gamma, \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2) = (\theta_2 \circ \theta_1, \tau_2) \\
&\quad \text{where} \\
&\qquad (\theta_1, \tau_1) = \mathsf{W}(\Gamma, e_1) \\
&\qquad \sigma = \mathsf{generalise}(\theta_1\ \Gamma, \tau_1) \\
&\qquad (\theta_2, \tau_2) = \mathsf{W}(\theta_1(\Gamma, x :: \sigma), e_2)
\end{aligned}
$$

The function **generalise** is as before: it looks for all inference variables that occur free in x but not in Γ and quantifies over them.

Unification for variables

$$\begin{aligned}
&\mathsf{unify}(\alpha, \alpha) = \mathsf{id} \\
&\mathsf{unify}(\alpha, \tau) = [\alpha \mapsto \tau] \\
&\quad \text{where} \\
&\qquad \alpha \text{ does not occur free in } \tau \\
&\mathsf{unify}(\tau, \alpha) = \mathsf{unify}(\alpha, \tau)
\end{aligned}$$

These are the interesting cases: a variable can be unified with a monotype by substituting one for the other.

Unification for variables

$$\begin{aligned}
&\mathsf{unify}(\alpha, \alpha) = \mathsf{id} \\
&\mathsf{unify}(\alpha, \tau) = [\alpha \mapsto \tau] \\
&\quad \text{where} \\
&\qquad \alpha \text{ does not occur free in } \tau \\
&\mathsf{unify}(\tau, \alpha) = \mathsf{unify}(\alpha, \tau)
\end{aligned}$$

These are the interesting cases: a variable can be unified with a monotype by substituting one for the other.

The restriction prevents infinite types.

Unification for functions

$$\textsf{unify}(\tau_1 \to \tau_2, \tau_1' \to \tau_2') = \theta_2 \circ \theta_1$$

where

$$\theta_1 = \textsf{unify}(\tau_1, \tau_1')$$

$$\theta_2 = \textsf{unify}(\theta_1\tau_2, \theta_1\tau_2')$$

A function type unifies with another function type if both the arguments and the results unify.

Unification for type constructors

$$\textsf{unify}(T_n\ \tau_1 \ldots \tau_n, T_n\ \tau'_1 \ldots \tau'_n) = \theta_n \circ \cdots \circ \theta_1$$

where

$$\theta_1 = \textsf{unify}(\tau_1, \tau'_1)$$

$$\ldots$$

$$\theta_n = \textsf{unify}((\theta_{n-1} \circ \cdots \circ \theta_1)\tau_n, (\theta_{n-1} \circ \cdots \circ \theta_1)\tau'_n)$$

A type constructor application unifies with another application of the same type constructor if all the arguments unify pointwise.

Unification for type constructors

$$\textsf{unify}(T_n\ \tau_1 \ldots \tau_n, T_n\ \tau'_1 \ldots \tau'_n) = \theta_n \circ \cdots \circ \theta_1$$

where

$$\theta_1 = \textsf{unify}(\tau_1, \tau'_1)$$

$$\ldots$$

$$\theta_n = \textsf{unify}((\theta_{n-1} \circ \cdots \circ \theta_1)\tau_n, (\theta_{n-1} \circ \cdots \circ \theta_1)\tau'_n)$$

A type constructor application unifies with another application of the same type constructor if all the arguments unify pointwise.

All other cases of unification fail.

Properties of Algorithm W

Soundness

If $\mathsf{W}(\Gamma, e) = (\theta, \tau)$, then $\theta\Gamma \vdash e :: \tau$ is derivable from the type rules.

Completeness / principal types

If a type for e in Γ is derivable, then $\mathsf{W}(\Gamma, e)$ succeeds and (after generalisation) yields the principal type of e.

Principal types

A (poly)type is called an instance of another (poly)type if it can be obtained by instantiating and re-generalising some of its type variables.

Principal types

A (poly)type is called an instance of another (poly)type if it can be obtained by instantiating and re-generalising some of its type variables.

Example:

$$\forall \alpha \beta.\ \alpha \to \beta \to \alpha$$

Instances:

$$\forall \alpha.\ \alpha \to \alpha \to \alpha$$
$$\forall \alpha \beta.\ [\alpha] \to \beta \to [\alpha]$$
$$\forall \alpha.\ \alpha \to \texttt{Int} \to \alpha$$
$$\texttt{Bool} \to \texttt{Int} \to \texttt{Bool}$$
$$\texttt{Int} \to \texttt{Int} \to \texttt{Int}$$

Principal types (contd.)

A polytype σ is called the principal type of an expression if all valid types of the expression are instances of σ.

Implementation

Demo

Extensions

Pattern matching

As already discussed briefly, simple pattern matching can be reduced to the application of a recursive function:

```
caseList :: [a] -> r -> (a -> [a] -> r) -> r
caseList xs nil cons =
  case xs of
    []     -> nil
    y : ys -> cons y ys
```

Pattern matching

As already discussed briefly, simple pattern matching can be reduced to the application of a recursive function:

```
caseList :: [a] -> r -> (a -> [a] -> r) -> r
caseList xs nil cons =
  case xs of
    []     -> nil
    y : ys -> cons y ys
```

More complex patterns require slightly more disciplined extensions to the DHM, but don't add any significant difficulty.

Recursion

As said before, general recursion can be added to the system by adding `fix` or similar combinators to the initial environment.

Recursion

As said before, general recursion can be added to the system by adding `fix` or similar combinators to the initial environment.

It is also not significantly more difficult to add a recursive let construct such as Haskell offers to the language and infer types for it directly.

Type annotations and polymorphic recursion

In the context of nested datatypes, we discussed polymorphic recursion, and that polymorphically recursive functions require type signatures in Haskell.

How is this compatible with the completeness of Algorithm W?

Type annotations and polymorphic recursion (contd.)

- Polymorphically recursive functions have no derivable type in DHM.
- The reason is that if recursion is internally modelled via `fix`, a recursive call corresponds to referring to a lambda-bound variable.
- For lambda-bound variables, we assume a type variable, and that type variable can be substituted for one fixed monotype.

Type annotations and polymorphic recursion (contd.)

- It is easy to extend DHM with the possibility of providing explicit type annotations.
- Recursive let bindings can be annotated with explicit polytypes.
- It is then easy to check such a binding even if it uses polymorphic recursion: we simply extend the environment with the given type for both the declaration and the body. We then also have to check that the annotation is correct.
- Similarly, Haskell allows annotations everywhere (that are then checked for correctness).

Datatypes

A `data` declaration:

- introduces a new type constructor,
- introduces new constructor functions (can be treated as functions in the initial environment),
- introduces new patterns for pattern matching (see previous slide).

Syntactic sugar

Many constructs are just syntactic sugar and can easily be reduced to the simple language features of `DHM` and built-in functions / type constructors:

- operator sections,
- `if` - `then` - `else`,
- guards,
- `where` clauses,
- list enumeration syntax,
- list comprehensions,
- do notation
- ...

Type classes

A very significant extension to DHM is the presence of type classes in Haskell:

- This requires extending the algorithm to not just be parameterized on a type environment, but also a context of assumed class instances on locally known type variables.
- In addition, `class` and `instance` declarations determine an entailment relation that allows simplifying class contexts.

Kinds

Another significant extension is that DHMs assumption that all type constructors have a fixed arity is lifted in Haskell:

- Haskell has a kind system which is yet another layer of types, one level up.
- A similar system to DHM can be used yet again to perform kind inference.
- Nevertheless, there are some subtle interactions between the two layers.

More significant extensions

Haskell has many more advanced type system features that require more significant extensions to DHM (and are generally no longer compatible with inference, i.e., require more explicit annotations):

- Generalised algebraic datatypes
- Higher-rank polymorphism
- Multi-parameter type classes, functional dependencies
- Type families
- ...

Despite all these extensions, the ideas of DHM are still at the core of the type inference engine.