Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Advanced Functional Programming

2012-2013, periode 2

Doaitse Swierstra

Department of Information and Computing Sciences
Utrecht University

December 20, 2010

11. Types and type classes

This lecture

Types and type classes

- Prerequisites
- Type inference
- Introduction to type classes
- Qualified types
- Evidence translation
- Defaulting
- Extensions

11.1 Prerequisites

Type checking vs. type inference

Type checking

Given a type-annotated program, decide whether the program is correctly typed.

Type checking vs. type inference

Type checking

Given a type-annotated program, decide whether the program is correctly typed.

Type inference

Given an un-annotated program, recover all the type annotations such that the annotated program is correctly typed.

Types and free variables

Question

How do we assign a type to a term with free variables?

> λx . plus x one

Types and free variables

Question

How do we assign a type to a term with free variables?

> λx . plus x one

Answer

We cannot unless we know the types of the free variables.

Environments

We therefore do not assign types to terms, but types to terms in a certain **environment** (also called **context**).

Environments

$$\begin{array}{lll} \Gamma ::= \varepsilon & & \text{empty environment} \\ \quad \mid & \Gamma, \mathsf{x} : \tau & \text{binding} \end{array}$$

Later bindings for a variable always shadow earlier bindings.

The typing relation

A statement of the form

$$\Gamma \vdash \mathsf{e} : \tau$$

can be read as follows:

In environment Γ, term e has type τ.

The typing relation

A statement of the form

$$\Gamma \vdash \mathsf{e} : \tau$$

can be read as follows:

In environment Γ, term e has type τ.

Note that $\Gamma \vdash \mathsf{e} : \tau$ is formally a ternary **relation** between an environment, a term and a type.

The $\vdash$ (called turnstile) and the colon are just notation for making the relation look nice but carry no meaning. We could have chosen the notation $\mathsf{T}\ (\Gamma, \mathsf{e}, \tau)$ for the relation as well, but $\Gamma \vdash \mathsf{e} : \tau$ is commonly used.

Type rules

The relation is defined inductively, using **inference rules**.

Type rules

The relation is defined inductively, using **inference rules**.

Variables

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

Type rules

The relation is defined inductively, using **inference rules**.

Variables

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

Above the bar are the **premises**.

Below the bar is the **conclusion**.

If the premises hold, we can infer the conclusion.

11.2 Type inference

Damas-Milner type inference

(Also called Hindley-Milner type inference.)

Mainly based on a paper by Milner (1978).

This algorithm is:

Damas-Milner type inference

(Also called Hindley-Milner type inference.)

Mainly based on a paper by Milner (1978).

This algorithm is:

- the basis of the algorithm used for the ML family of languages as well as Haskell;

Damas-Milner type inference

(Also called Hindley-Milner type inference.)

Mainly based on a paper by Milner (1978).

This algorithm is:

- the basis of the algorithm used for the ML family of languages as well as Haskell;
- allows type inference essentially for the simply-typed lambda calculus extended with a limited form of polymorphism (sometimes called **let**-polymorphism);

Damas-Milner type inference

(Also called Hindley-Milner type inference.)

Mainly based on a paper by Milner (1978).

This algorithm is:

- the basis of the algorithm used for the ML family of languages as well as Haskell;
- allows type inference essentially for the simply-typed lambda calculus extended with a limited form of polymorphism (sometimes called **let**-polymorphism);
- is a "sweet spot" in the design space: some simple extensions are possible (and performed), but fundamental extensions are typically significantly more difficult.

Monotypes and type schemes

Damas-Milner types can be polymorphic only on the outside.

That is why Haskell typically does not use an explicit universal quantifier.

Monotypes

Monotypes τ are types built from variables and type constructors.

Type schemes (or polytypes)

$$\begin{array}{lll} \sigma ::= \tau & & \text{monotype} \\ \quad \mid & \forall \alpha . \sigma & \text{quantified type} \end{array}$$

The key idea

The Damas-Milner algorithm distinguishes lambda-bound and let-bound (term) variables:

- lambda-bound variables are always assumed to have a monotype;
- of let-bound variables, we know what they are bound to, therefore they can have polymorphic type.

Inference variables

Whenever a lambda-bound variable is encountered, a fresh inference variable is introduced.

The variable represents a monotype.

When we learn more about the types, inference variables can be substituted by types.

Inference variables are different from universally quantified variables that express polymorphism.

Term language

e ::=	x	variables
	e e	application
	λx . e	abstraction
	let x = e **in** e	let binding

Only a simple language to start with, but we include **let** compared to plain lambda calculus.

Example

Assume an environment $\Gamma \equiv \mathsf{neg} : \mathsf{Nat} \to \mathsf{Nat}$.

Example

Assume an environment $\Gamma \equiv \text{neg} : \text{Nat} \to \text{Nat}$.

Consider λx . neg x.

Example

Assume an environment $\Gamma \equiv \text{neg} : \text{Nat} \to \text{Nat}$.

Consider λx . neg x.

For x, we introduce an inference variable v and assume $x : v$.

Example

Assume an environment $\Gamma \equiv \text{neg} : \text{Nat} \to \text{Nat}$.

Consider λx . neg x.

For x, we introduce an inference variable v and assume $x : v$.

To typecheck neg x, we first determine the types of the components.

Example

Assume an environment $\Gamma \equiv \text{neg} : \text{Nat} \to \text{Nat}$.

Consider λx . neg x.

For x, we introduce an inference variable v and assume $x : v$.

To typecheck neg x, we first determine the types of the components.

From the environment we learn $\text{neg} : \text{Nat} \to \text{Nat}$ and $x : v$.

Example

Assume an environment $\Gamma \equiv \text{neg} : \text{Nat} \to \text{Nat}$.

Consider λx . neg x.

For x, we introduce an inference variable v and assume $x : v$.

To typecheck neg x, we first determine the types of the components.

From the environment we learn $\text{neg} : \text{Nat} \to \text{Nat}$ and $x : v$.

We now unify Nat and v, introducing the substitution $v \mapsto \text{Nat}$.

Generalization and instantiation

Consider

$$\begin{array}{l}\textbf{let}\ \mathsf{id} = \lambda \mathsf{x}\ .\ \mathsf{x} \\ \textbf{in}\ \ (\mathsf{id}\ \mathsf{False}, \mathsf{id}\ \texttt{'x'})\end{array}$$

Generalization and instantiation

Consider

> **let** id $= \lambda x \,.\, x$
> **in** (id False, id 'x')

Inference for $\lambda x \,.\, x$ gives us the type $v \to v$ for some inference variable v, and there are no further assumptions about v.

Generalization and instantiation

Consider

> **let** id $= \lambda x\ .\ x$
> **in** (id False, id 'x')

Inference for $\lambda x\ .\ x$ gives us the type $v \to v$ for some inference variable v, and there are no further assumptions about v.

On a let-binding, the algorithm generalizes the inferred type as much as possible, in this case to $id : \forall a.a \to a$.

Generalization and instantiation

Consider

> **let** id $= \lambda x \,.\, x$
> **in** (id False, id 'x')

Inference for $\lambda x \,.\, x$ gives us the type $v \to v$ for some inference variable v, and there are no further assumptions about v.

On a let-binding, the algorithm generalizes the inferred type as much as possible, in this case to $\mathrm{id} : \forall a.a \to a$.

For every use, a polymorphic type is instantiated with fresh inference variables. For example, we get $w \to w$ for the first call, $u \to u$ for the second.

Generalization and instantiation

Consider

> **let** id $= \lambda x\ .\ x$
> **in** (id False, id 'x')

Inference for $\lambda x\ .\ x$ gives us the type $v \to v$ for some inference variable v, and there are no further assumptions about v.

On a let-binding, the algorithm generalizes the inferred type as much as possible, in this case to $id : \forall a.a \to a$.

For every use, a polymorphic type is instantiated with fresh inference variables. For example, we get $w \to w$ for the first call, $u \to u$ for the second.

The w gets unified with Bool, and u with Char.

Generalization again

Not everything can be generalized – assume that singleton : $\forall a.a \to [a]$

$$\lambda x \,.\, \textbf{let}\ y = \text{singleton}\ x \ \textbf{in}\ \text{head}\ y$$

Generalization again

Not everything can be generalized – assume that singleton : $\forall a.a \to [a]$

$$\lambda x \,.\, \textbf{let}\ y = \text{singleton}\ x \ \textbf{in}\ \text{head}\ y$$

For x, an inference variable v is introdued.

Generalization again

Not everything can be generalized – assume that singleton : $\forall a.a \to [a]$

$$\lambda x \,.\, \textbf{let}\ y = \text{singleton}\ x \ \textbf{in}\ \text{head}\ y$$

For x, an inference variable v is introdued.

Consequently, we infer the type $[v]$ for singleton x.

Generalization again

Not everything can be generalized – assume that singleton : $\forall a.a \to [a]$

$$\lambda x \,.\, \textbf{let}\ y = \text{singleton}\ x \\ \quad\ \ \textbf{in}\ \ \text{head}\ y$$

For x, an inference variable v is introdued.

Consequently, we infer the type $[v]$ for singleton x.

But we must not generalize the type of y to $\forall a.[a]$.

Generalization again

Not everything can be generalized – assume that singleton : $\forall a.a \to [a]$

$$\lambda x \,.\, \textbf{let}\ y = \text{singleton}\ x \ \textbf{in}\ \text{head}\ y$$

For x, an inference variable v is introdued.

Consequently, we infer the type $[v]$ for singleton x.

But we must not generalize the type of y to $\forall a.[a]$.

We can only generalize if a variable is not mentioned in the environment.

Motivation: unification

Question

What is the type of the following expressions?

> $\lambda x\ y \rightarrow$ 'a'
> $\lambda x\ y \rightarrow$ **if** x **then** y **else** y
> $[\lambda x\ y \rightarrow$ 'a', $\lambda x\ y \rightarrow$ **if** x **then** y **else** y$]$

Unification

Given two types that contain inference variables, a **unification** of the two types is a substitution on inference variables that makes both types equal.

Unification

Given two types that contain inference variables, a **unification** of the two types is a substitution on inference variables that makes both types equal.

> $[\lambda x\ y \to \text{'a'}, \lambda x\ y \to \textbf{if}\ x\ \textbf{then}\ y\ \textbf{else}\ y]$

We have to unify the two types

> $v \to w \to \text{Char}$
> $\text{Bool} \to u \to u$

Unification

Given two types that contain inference variables, a **unification** of the two types is a substitution on inference variables that makes both types equal.

> $[\lambda x\ y \to \text{'a'}, \lambda x\ y \to \textbf{if}\ x\ \textbf{then}\ y\ \textbf{else}\ y]$

We have to unify the two types

> $v \to w \to Char$
> $Bool \to u \to u$

> $u \mapsto Char, w \mapsto Char, v \mapsto Bool$

Unification – contd.

What if we want to unify the following types:

$$v \to w \to \text{Char}$$
$$v \to w \to u$$

Unification – contd.

What if we want to unify the following types:

$$v \to w \to \text{Char}$$
$$v \to w \to u$$

What about the substitution:

$$v \mapsto w, u \mapsto \text{Char}$$

Unification – contd.

What if we want to unify the following types:

$$v \to w \to \text{Char}$$
$$v \to w \to u$$

What about the substitution:

$$v \mapsto w, u \mapsto \text{Char}$$

We are interested in the **minimal** substitution.

Unification – contd.

What if we want to unify the types:

> w
> $v \to u$

Unification – contd.

What if we want to unify the types:

> w
> $v \to u$

And how about

> u
> $u \to u$

Unification – contd.

What if we want to unify the types:

> w
> $v \to u$

And how about

> u
> $u \to u$

A substitution $u \mapsto u \to u$ would result in an infinite type. Most systems (including Haskell) reject infinite types, and make this a type error.

Idea of the unification algorithm

We distinguish the following cases:

Idea of the unification algorithm

We distinguish the following cases:

- if we have two equal inference variables, then there is nothing to do;

Idea of the unification algorithm

We distinguish the following cases:

- if we have two equal inference variables, then there is nothing to do;
- if we have an inference variable and another type that does not contain the inference variable (**occurs check** to prevent infinite types), we substitute the variable by the other type;

Idea of the unification algorithm

We distinguish the following cases:

- if we have two equal inference variables, then there is nothing to do;
- if we have an inference variable and another type that does not contain the inference variable (**occurs check** to prevent infinite types), we substitute the variable by the other type;
- if we have two function types, we recursively unify the domains and codomains;
- if we have two equal type variables, there is nothing to do;
- if we have any other situation, unification fails.

Principal types

There is a similar notion for types as we had for unifications. One type can be more general than another:

$$
\begin{array}{ll}
a & \rightarrow b \\
(a, b) & \rightarrow (b, a) \\
(a, a) & \rightarrow (a, a) \\
(Int, Int) & \rightarrow (Int, Int)
\end{array}
$$

Principal types

There is a similar notion for types as we had for unifications. One type can be more general than another:

$$
\begin{array}{ll}
a & \to b \\
(a, b) & \to (b, a) \\
(a, a) & \to (a, a) \\
(\mathsf{Int}, \mathsf{Int}) & \to (\mathsf{Int}, \mathsf{Int})
\end{array}
$$

Damas-Milner type inference always infers the most general type (called the **principal type**).

What is missing?

- Top-level declarations.
- Mutually recursive definitions.
- Explicit type annotations.
- Kinds.
- Datatypes and pattern matching.
- Type classes.
- ...

Type classes

- One of the features that makes Haskell 'unique'.
- Predicates on types (but not types themselves!).
- Provide ad-hoc polymorphism or overloading.
- Extensible or open.
- Haskell 98 only allows unary predicates, but already allows classes that range over types of different kinds (Eq and Show vs. Functor and Monad).
- Lots of extensions.
- Can be translated into polymorphic lambda calculus F_ω.

11.3 Introduction to type classes

Classes and instances

- A **class** declaration defines a predicate. Each member of a class supports a certain set of methods.
- An **instance** declaration declares some types to be in the class, and provides evidence of that fact by providing implementations for the methods.
- Depending on the situation, we may ask different questions about a type and a class:
  - Is the type a member of the class (yes or no)?
  - Why/how is the type a member of the class (give me evidence, please)?
- Functions that use methods get class constraints that are like proof obligations.

Parametric vs. ad-hoc polymorphism

Parametric polymorphism

$$\begin{aligned}&\text{swap} :: (a, b) \to (b, a)\\&\text{swap } (x, y) = (y, x)\end{aligned}$$

Unconstrained variables can be instantiated to all types. No assumptions about the type can be made in the definition of the function. The function works uniformly for all types.

Ad-hoc polymorphism

$$\begin{aligned}&\text{between} :: (\text{Ord } a) \Rightarrow a \to a \to a \to \text{Bool}\\&\text{between } x\ y\ z = x \leqslant y \wedge y \leqslant z\end{aligned}$$

Constrained variables can only be instantiated to members of the class. Since each instance is specific to a type, the behaviour can differ vastly depending on the type that is used.

Restrictions

The Haskell 98 design is rather restrictive:

- only one type parameter per class
- only one instance per type
- superclasses are possible, but the class hierarchy must not be cyclic
- instances can only be declared for simple types, types of the form $T\ a_1 \ldots a_n$ (where T is not a type synonym and $a_1, \ldots, a_n$ are type variables).
- instance or class contexts may only be of the form C a (where a is a type variable).
- function contexts can only be of the form $C\ (a\ t_1 \ldots t_n)$ (where a is a type variable and $t_1, \ldots, t_n$ are types possibly containing variables).

Examples: Restrictions

```
instance Eq a    ⇒ Eq [a]
instance Eq [a]  ⇒ Eq [a]              -- illegal
instance Eq Int  ⇒ Eq [Int]            -- illegal
instance Eq a    ⇒ Eq (a, Bool)        -- illegal
instance Eq [[a]]                      -- illegal
instance Eq String                     -- illegal

(Eq (f Int)) ⇒ f Int → f Int → Bool
(Eq [Int])   ⇒ [Int] → [Int] → Bool    -- illegal
(Eq [a])     ⇒ [a]   → [a]   → Bool    -- illegal
```

Examples: Restrictions

```
instance Eq a    ⇒ Eq [a]
instance Eq [a]  ⇒ Eq [a]              -- illegal
instance Eq Int  ⇒ Eq [Int]            -- illegal
instance Eq a    ⇒ Eq (a, Bool)        -- illegal
instance Eq [[a]]                      -- illegal
instance Eq String                     -- illegal

(Eq (f Int)) ⇒ f Int → f Int → Bool
(Eq [Int])   ⇒ [Int] → [Int] → Bool    -- illegal
(Eq [a])     ⇒ [a]   → [a]   → Bool    -- illegal
```

The restrictions ensure that instance resolution is efficient and terminates, and that contexts are always reduced as much as possible.

11.4 Qualified types

Introduction

- Types with contexts are also called qualified types.
- Mark Jones describes a Theory of Qualified Types, which is a framework of which the Haskell type class system is one specific instance.
- Qualified types can also be used to track other properties of types:
  - presence or absence of labels in extensible records,
  - subtyping conditions
  - type equality constraints
  - presence or absence of effects (see Hage, Holdermans, Middelkoop, ICFP 2007)

Example

Some contexts imply other contexts:

$$
\begin{array}{ll}
\mathsf{Eq\ Int} & \Vdash \mathsf{Eq\ [Int]} \\
\mathsf{Eq\ Bool}, \mathsf{Ord\ Int} & \Vdash \mathsf{Ord\ Int} \\
\emptyset & \Vdash \mathsf{Eq\ Int}
\end{array}
$$

The latter holds if we assume globally that an instance for Eq Int exists.

Entailment

Entailment ($\Vdash$) is a relation on two contexts, i.e., between two sets.

We assume that the following property (set-entails) holds:

$$P \Vdash Q \quad \text{if and only if} \quad \text{for all } \pi \text{ in q, } P \Vdash \pi$$

Basic entailment rules

The following rules are given by the framework for qualified types:

$$\frac{Q \subseteq P}{P \Vdash Q} \text{ (mono)}$$

$$\frac{P \Vdash Q \quad Q \Vdash R}{P \Vdash R} \text{ (trans)}$$

$$\frac{P \Vdash Q \quad \varphi \text{ is a substitution}}{\varphi\, P \Vdash \varphi\, Q} \text{ (closure)}$$

Derived rules

Some properties can easily be derived from the previous rules.

Directly from (mono):

$$\frac{}{P \Vdash P}\ (\text{id}) \qquad \frac{}{P \Vdash \emptyset}\ (\text{term})$$

$$\frac{}{P, Q \Vdash P}\ (\text{fst}) \qquad \frac{}{P, Q \Vdash Q}\ (\text{snd})$$

From (mono) and (set-entails):

$$\frac{P \Vdash Q \quad P \Vdash R}{P \Vdash Q, R}\ (\text{univ})$$

Derived rules – contd.

Using these rules, we can derive yet more complex (but still widely useful) rules:

$$\frac{P \Vdash Q \quad P' \Vdash Q'}{P, P' \Vdash Q, Q'} \text{ (dist)}$$

Derived rules – contd.

Using these rules, we can derive yet more complex (but still widely useful) rules:

$$\frac{P \Vdash Q \quad P' \Vdash Q'}{P, P' \Vdash Q, Q'}\ (\text{dist})$$

Proof

$$\dfrac{\dfrac{\dfrac{}{P, P' \Vdash P}\ (\text{mono}) \quad \dfrac{}{P \Vdash Q}}{P, P' \Vdash Q}\ (\text{trans}) \quad \dfrac{\dfrac{}{P, P' \Vdash P'}\ (\text{mono}) \quad \dfrac{}{P' \Vdash Q'}}{P, P' \Vdash Q'}\ (\text{trans})}{P, P' \Vdash Q, Q'}\ (\text{univ})$$

Type class entailment

Only these two rules are specific to the type class system:

$$\frac{P \Vdash \pi \quad \textbf{class}\ Q \Rightarrow \pi}{P \Vdash Q}\ (\text{super})$$

$$\frac{P \Vdash Q \quad \textbf{instance}\ Q \Rightarrow \pi}{P \Vdash \pi}\ (\text{inst})$$

Type class entailment

Only these two rules are specific to the type class system:

$$\frac{P \Vdash \pi \quad \textbf{class}\ Q \Rightarrow \pi}{P \Vdash Q}\ (\text{super})$$

$$\frac{P \Vdash Q \quad \textbf{instance}\ Q \Rightarrow \pi}{P \Vdash \pi}\ (\text{inst})$$

Example

$$\frac{\textbf{class}\ \text{Eq a} \Rightarrow \text{Ord a}}{\text{Ord a} \Vdash \text{Eq a}}\ (\text{super})$$

The direction of the arrow is somewhat misleading. Not Eq a implies Ord a, but the other way around. Read: "only if Eq a, we can define Ord a".

Validity of instances

Instance declarations must adhere to the class hierarchy:

$$\frac{\textbf{class}\ \mathsf{Q} \Rightarrow \pi \quad \mathsf{P} \Vdash \varphi\ \mathsf{Q}}{\textbf{instance}\ \mathsf{P} \Rightarrow \varphi\ \pi\ \text{is valid}}\ (\text{valid})$$

Example: validity of instances

```
class (Eq a, Show a) ⇒ Num a
class Foo a ⇒ Bar a
class Foo a

instance (Eq a, Show a) ⇒ Foo [a]
instance Num a          ⇒ Bar [a]
```

Example: validity of instances

```
class (Eq a, Show a) ⇒ Num a
class Foo a ⇒ Bar a
class Foo a

instance (Eq a, Show a) ⇒ Foo [a]
instance Num a          ⇒ Bar [a]
```

$$\dfrac{\dfrac{}{\mathbf{c}\ \mathsf{Foo\ a} \Rightarrow \mathsf{Bar\ a}} \quad \dfrac{\dots}{\mathsf{Num\ a} \Vdash \mathsf{Foo}\ [\mathsf{a}]}}{\mathbf{i}\ \mathsf{Num\ a} \Rightarrow \mathsf{Bar}\ [\mathsf{a}]\ \text{is valid}}\ (\text{valid})$$

Example: validity of instances

```
class (Eq a, Show a) ⇒ Num a
class Foo a ⇒ Bar a
class Foo a

instance (Eq a, Show a) ⇒ Foo [a]
instance Num a          ⇒ Bar [a]
```

$$\frac{\dfrac{}{\mathbf{c}\ \mathsf{Foo\ a} \Rightarrow \mathsf{Bar\ a}} \quad \dfrac{\dfrac{\cdots}{}}{\mathsf{Num\ a} \Vdash \mathsf{Foo}\ [\mathsf{a}]}}{\mathbf{i}\ \mathsf{Num\ a} \Rightarrow \mathsf{Bar}\ [\mathsf{a}]\ \mathsf{is\ valid}}\ (\mathsf{valid})$$

$$\frac{\dfrac{\dfrac{}{\mathbf{c}\ (\mathsf{Eq\ a}, \mathsf{Show\ a}) \Rightarrow \mathsf{Num\ a}}\ (\mathsf{class})}{\mathsf{Num\ a} \Vdash (\mathsf{Eq\ a}, \mathsf{Show\ a})} \quad \dfrac{}{\mathbf{i}\ (\mathsf{Eq\ a}, \mathsf{Show\ a}) \Rightarrow \mathsf{Foo}\ [\mathsf{a}]}}{\mathsf{Num\ a} \Vdash \mathsf{Foo}\ [\mathsf{a}]}\ (\mathsf{inst})$$

Type rules

Usually, type rules are of the form

$$\Gamma \vdash \mathsf{e} :: \tau$$

where Γ is an environment mapping identifiers to types, e is an expression, and t is a (possibly polymorphic) type.

Type rules

Usually, type rules are of the form

$$\Gamma \vdash \mathsf{e} :: \tau$$

where Γ is an environment mapping identifiers to types, e is an expression, and t is a (possibly polymorphic) type.

With qualified types, type rules are of the form

$$\mathsf{P} \mid \Gamma \vdash \mathsf{e} :: \tau$$

where P is a context representing (local) knowledge, and τ is a (possibly polymorphic, possibly overloaded) type.

Context reduction

$$\frac{\mathsf{P} \mid \Gamma \vdash \mathsf{e} :: \pi \Rightarrow \rho \quad \mathsf{P} \Vdash \pi}{\mathsf{P} \mid \Gamma \vdash \mathsf{e} :: \rho} \text{ (context-reduce)}$$

Haskell's type inference applies this rule where adequate:

```
(==)                    :: (Eq a) ⇒ a → a → Bool
"hello"                 :: String
"hello" == "hello"      :: Bool
```

Requires $\emptyset \Vdash$ Eq String.

Context introduction

$$\frac{P, \pi \mid \Gamma \vdash e :: \rho}{P \mid \Gamma \vdash e :: \pi \Rightarrow \rho} \text{ (context-intro)}$$

Haskell's type inference applies this rule when generalizing in a **let** or a toplevel declaration:

> between x y z = x ⩽ y ∧ y ⩽ z

Inferred to be of type (Ord a) ⇒ a → a → a → Bool.

A strange error

Using the following definition in a Haskell module results in a type error:

```
maxList = maximum
```

A strange error

Using the following definition in a Haskell module results in a type error:

```
maxList = maximum
```

Monomorphism restriction

A toplevel value without an explicit type signature is never overloaded.

11.5 Evidence translation

Translating type classes

Type classes can be translated into a lambda calculus without type classes as follows:

- Each class declaration defines a record type (also called dictionary).
- Each instance declaration defines a function resulting in the dictionary type.
- Each method call selects the corresponding field from the dictionary.
- Context introduction corresponds to the abstraction of function arguments of dictionary type.
- Context reduction corresponds to the implicit construction and application of a dictionary argument.

Example: evidence translation

```
class E a where
  e :: a → a → Bool

instance E Int where
  e = (==)


instance E a ⇒ E [a] where
  e []       []       = True
  e (x : xs) (y : ys) = e x y ∧ e xs ys
  e _        _        = False

member :: E a ⇒ a → [a] → Bool
member _ []       = False
member a (x : xs) = e a x ∨ member a xs

duplicates :: E a ⇒ [a] → Bool
duplicates []       = False
duplicates (x : xs) =
  member x xs ∨ duplicates xs

is = [[], [1], [2], [1, 2], [2, 1]] :: [[Int]]

main = (duplicates is,
        duplicates (concat is))
```

Example: evidence translation

```
class E a where
  e :: a → a → Bool

instance E Int where
  e = (==)

instance E a ⇒ E [a] where
  e []     []     = True
  e (x:xs) (y:ys) = e x y ∧ e xs ys
  e _      _      = False

member :: E a ⇒ a → [a] → Bool
member _ []     = False
member a (x:xs) = e a x ∨ member a xs

duplicates :: E a ⇒ [a] → Bool
duplicates []     = False
duplicates (x:xs) =
  member x xs ∨ duplicates xs

is = [[], [1], [2], [1,2], [2,1]] :: [[Int]]

main = (duplicates is,
        duplicates (concat is))
```

```
data E a = E
  {e :: a → a → Bool}

e_Int :: E Int
e_Int = E {e = (==)}

e_List :: E a → E [a]
e_List e_a = E {e = e'} where
  e' []     []     = True
  e' (x:xs) (y:ys) = e e_a x y ∧ e (e_List e_a) xs ys
  e' _      _      = False

member :: E a → a → [a] → Bool
member e_a _ []     = False
member e_a a (x:xs) = e e_a a x ∨ member e_a a xs

duplicates :: E a → [a] → Bool
duplicates e_a []     = False
duplicates e_a (x:xs) =
  member e_a x xs ∨ duplicates e_a xs

is = [[], [1], [2], [1,2], [2,1]] :: [[Int]]

main = (duplicates (e_List e_Int) is,
        duplicates e_Int (concat is))
```

Dictionaries and superclasses

Dictionaries contain dictionaries of their superclasses:

```
class Eq a ⇒ Ord a
```

```
data Ord a = Ord
  {eq      :: Eq a,
   compare :: a → a → Ordering,
   ...}
```

Dictionaries and polymorphic methods

```
class Functor f where
  fmap :: (a → b) → f a → f b
```

```
data Functor f = Functor
  {fmap :: ∀a b.(a → b) → f a → f b}
```

The field fmap is a polymorphic field. Note that this is equivalent to the non-record

```
data Functor f = Functor (∀a b.(a → b) → f a → f b)
```

and different from the existential type

```
data Functor' f = ∀a b.Functor' ((a → b) → f a → f b)
```

Polymorphic fields vs. existential types

- An existential type hides a specific type.
- A polymorphic field stores a polymorphic function.

```
data Functor  f = Functor (∀a b.(a → b) → f a → f b)
data Functor' f = ∀a b.Functor' ((a → b) → f a → f b)
Functor  :: (∀a b.(a → b) → f a → f b) → Functor f
Functor' :: ∀a b.((a → b) → f a → f b) → Functor f
```

The constructor Functor takes a polymorphic function as an argument. The type of Functor is a so-called rank-2 polymorphic type. More in the next lecture.

Type-directed translation

Evidence translation is a byproduct of type inference – type rules can be augmented with translated terms. New form of rules:

$$\mathsf{P} \mid \Gamma \vdash \mathsf{e} \rightsquigarrow \mathsf{e}' :: \tau$$
$$\mathsf{P} \Vdash \mathsf{Q} \rightsquigarrow \mathsf{e}$$

Modified rules:

$$\frac{\mathsf{P} \mid \Gamma \vdash \mathsf{e} \rightsquigarrow \mathsf{e}' :: \pi \Rightarrow \rho \quad \mathsf{P} \Vdash \pi \rightsquigarrow \mathsf{e}_\pi}{\mathsf{P} \mid \Gamma \vdash \mathsf{e} \rightsquigarrow \mathsf{e}'\ \mathsf{e}_\pi :: \rho} \text{ (context-reduce)}$$

$$\frac{\mathsf{P}, \mathsf{e}_\pi :: \pi \mid \Gamma \vdash \mathsf{e} \rightsquigarrow \mathsf{e}' :: \rho}{\mathsf{P} \mid \Gamma \vdash \mathsf{e} \rightsquigarrow \lambda \mathsf{e}_\pi \to \mathsf{e}' :: \pi \Rightarrow \rho} \text{ (context-intro)}$$

Monomorphism restriction revisited

Question

Why the monomorphism restriction?

Monomorphism restriction revisited

Question

Why the monomorphism restriction?

Answer

To prevent unexpected inefficiency or loss of sharing.

11.6 Defaulting

Defaulting of numeric classes

> Main⟩ : t 42
> 42 :: (Num t) ⇒ t

Defining

> x = 42

does not produce an error despite the monomorphism restriction.

Defaulting of numeric classes

> Main⟩ : t 42
> 42 :: (Num t) ⇒ t

Defining

> x = 42

does not produce an error despite the monomorphism restriction.

Haskell performs defaulting of Num constraints and chooses Integer in this case.

GHCi defaulting

GHCi (not Haskell in general) also performs defaulting of other constraints than Num, to ():

```
Main⟩ let maxList = maximum
Main⟩  :t maxList
maxList :: [()] → ()
```

- Prevents annoying errors (show []).
- Can lead to subtle mistakes (QuickCheck properties).

Ambiguity and coherence

Question

What is the type of the following expression?

> show ∘ read

Ambiguity and coherence (contd.)

- Such an expression is called ambiguous because it has a constraint mentioning a variable that does not occur in the rest of the type:

  > $(\text{Show a}, \text{Read a}) \Rightarrow \text{String} \rightarrow \text{String}$

- Choosing different types can lead to different behaviour.
- Ambiguous types are disallowed by Haskell if they cannot be defaulted.
- Ambiguity is a form of incoherence: if allowed, multiple translations of a program with possibly different behaviour are possible.

Specialization

A specialization is a partially evaluated copy of an overloaded function – trades code size for more efficiency. GHC provides a pragma for this purpose:

```
between :: (Ord a) ⇒ a → a → a → Bool
{-# SPECIALISE between :: Char → Char → Char → Bool #-}
```

causes

$$\begin{aligned} &\text{between}_{\text{Char}} :: \text{Char} \to \text{Char} \to \text{Char} \to \text{Bool} \\ &\text{between}_{\text{Char}} = \text{between ord}_{\text{Char}} \end{aligned}$$

to be generated and used whenever between ord_{Char} would normally be used.

Using a `RULES` pragma, one can even provide different implementations for specific types. (Why useful?)

11.7 Extensions

Extensions to the class system

- Nearly all Haskell-98 restrictions to the class system can be lifted.
- The price: worse properties of the program, less predictability, worse error messages, partially unclear semantics and interactions, possible compiler bugs.
- Nevertheless, some extensions are useful, and it is important to explore the design space in order to find an optimum.

Flexible instances and contexts

- Lifts the restrictions on the shape of instances and contexts.

Overlapping instances

- Allows overlapping instance definitions such as

```
instance Foo a ⇒ Foo [a]
instance         Foo [Int]
```

- Two possibilities to construct Foo [Int] if

```
instance Foo Int
```

  is also given. Both possibilities might lead to different behaviour (incoherence).
- The most specific instance is chosen.

Overlapping instances and context reduction

What about

```
foo :: (Foo a) ⇒ a → a
test x xs = foo (x : xs)
```

?

Overlapping instances and context reduction

What about

```
foo :: (Foo a) ⇒ a → a
test x xs = foo (x : xs)
```

?

Reducing the context of test from Foo [a] to Foo a prevents Foo [Int] from being selected! Delay?

Incoherent instances

If you let GHC infer a type for test in

```
foo :: (Foo a) ⇒ a → a
test x xs = foo (x : xs)
```

you get $(\text{Foo}\ [a]) \Rightarrow a \rightarrow [a] \rightarrow [a]$, i.e., GHC tries to delay the decision.

Incoherent instances (contd.)

If you specify

> test :: Foo a ⇒ a → [a] → [a]

you get an error unless you tell GHC to allow incoherent instances.

Advice

With incoherent instances, it is very hard to predict how the instances are built. Allow incoherent instances only if you make sure that different ways to construct an instance have the same behaviour!

Undecidable instances

With undecidable instances, it is no longer required that context "reduction" actually reduces the context. The type checker may loop:

```
instance Foo [[a]] ⇒ Foo [a]
```