Types in Programming Languages (11)

Christian Urban

http://www4.in.tum.de/lehre/vorlesungen/types/WS0607/

Recap from last Week

- We had a look at the Curry-Howard correspondence

Types	$\Leftrightarrow$	Formulae
Typed Terms	$\Leftrightarrow$	Proof
Evaluation	$\Leftrightarrow$	Proof Normalisation
Typing Problem	$\Leftrightarrow$	Finding a Proof

- We had a look at the Polymorphic Lambda-Calculus - used to encode algebraic datatypes.

Motivation

- "Arithmetic, equality, showing a value as a string: three operations guaranteed to give language designers nightmares" from Odersky et al.
- Equality: there are types for which equality should be defined, for others it should not.
- ML has a special sort (or class) of equality types, i.e. types over which equality is defined.
- Type classes allow the user to define such classes.

Type Classes

- A type class is defined by the set of operations/methods that must be implemented for every type in the class.
- A type can be made a member of a type class using an instance declaration.
- Note the difference with classes in OO (classes there are types; type classes are not types—they are more like Java's interfaces).
- There is no access control in a type class (needs to be implemented using modules).

Problems

There are some problems with type classes

- a program cannot be assigned a meaning independent of its types
- type-safety (well-typed programs cannot *go wrong*) cannot be formulated for transition relations
- every phrase in a program has a most general/principle type

Can be solved in restricted systems; e.g. single parameter type classes.

Intuition

The intuition behind type classes is as follows

- equal is a function with type

  $$X \to X \to \mathrm{bool}$$

  but under the assumption that X is of type class EQ.

Intuition

The intuition behind type classes is as follows

- equal is a function with type

$$\forall X.\ X \rightarrow X \rightarrow \mathsf{bool}$$

  but under the assumption that X is of type class EQ.

Intuition

The intuition behind type classes is as follows

- equal is a function with type

$$\forall X.\ X \to X \to \mathsf{bool}$$

  but under the assumption that X is of type class EQ.

- $\forall X$ such that $X \in \mathsf{EQ}.\ X \to X \to \mathsf{bool}$

Intuition

The intuition behind type classes is as follows

- equal is a function with type

  $$\forall X.\ X \rightarrow X \rightarrow \mathsf{bool}$$

  but under the assumption that X is of type class EQ.
- $\forall X$ such that $X \in \mathsf{EQ}.\ X \rightarrow X \rightarrow \mathsf{bool}$
- $\forall X.\ X \in \mathsf{EQ} \Rightarrow X \rightarrow X \rightarrow \mathsf{bool}$

Intuition

The intuition behind type classes is as follows

- equal is a function with type

  $$\forall X.\ X \rightarrow X \rightarrow \mathsf{bool}$$

  but under the assumption that X is of type class EQ.
- $\forall X$ such that $X \in \mathsf{EQ}.\ X \rightarrow X \rightarrow \mathsf{bool}$
- $\forall X.\ \mathsf{EQ}(X) \Rightarrow X \rightarrow X \rightarrow \mathsf{bool}$

Intuition

The intuition behind type classes is as follows

- equal is a function with type

  $$\forall X.\ X \to X \to \mathsf{bool}$$

  but under the assumption that X is of type class EQ.
- $\forall X$ such that $X \in \mathsf{EQ}.\ X \to X \to \mathsf{bool}$
- $\forall X.\ \mathsf{EQ}(X) \Rightarrow X \to X \to \mathsf{bool}$
- "Types" will be of the form

  $$\text{some constraints} \Rightarrow T$$

Concrete Example

class EQ(X) where
    equal : $X \rightarrow X \rightarrow$ bool

inst equal : int $\rightarrow$ int $\rightarrow$ bool
    equal $=$ primitive_equal_over_ints

list_equal : (equal:$X \rightarrow X \rightarrow$ bool) $\Rightarrow [X] \rightarrow [X] \rightarrow$ bool
    list_equal [] [] = True
    list_equal (x:xs) (y:ys) = equal x y $\wedge$ list_equal xs ys

inst equal : (equal:$X \rightarrow X \rightarrow$ bool) $\Rightarrow [X] \rightarrow [X] \rightarrow$ bool
    equal $=$ list_equal

Syntax

- Types:

$$
\begin{array}{lcl}
T & ::= & X \\
 & | & T \to T \\
 & | & \text{bool}, \text{int}, [X], \ldots
\end{array}
$$

- Type-schemes:

$$
\begin{array}{lcl}
S & ::= & T \\
 & | & \forall X.C(X) \Rightarrow S
\end{array}
$$

- Constraints:

$$
C(X) \quad ::= \quad \{o : X \to T, \ldots
$$

where T can contain X

Syntax

- Terms:

$$\begin{array}{rcl} e & ::= & x \\ & \mid & e\ e \\ & \mid & \lambda x.e \\ & \mid & \text{let } x = e \text{ in } e \end{array}$$

- Programs:

$$p \quad ::= \quad e \mid \text{inst } o : S_T = e \text{ in } p$$

where S is type-scheme with the condition that T can't be a variable

Concrete Syntax

- For $\forall X.\, o : X \to T_1 \Rightarrow T_2$ we write

$$o : X \to T_1 \Rightarrow T_2$$

list_equal : (equal:$X \to X \to$ bool) $\Rightarrow [X] \to [X] \to$ bool

- For inst $o : S = e$ we write

$$o : S$$
$$o = e$$

inst equal : (equal:$X \to X \to$ bool) $\Rightarrow [X] \to [X] \to$ bool
    equal = list_equal

Type-System

$$\frac{\mathsf{valid}\,\Gamma \quad x : S \in \Gamma}{\Gamma \vdash x : S}$$

$$\frac{\Gamma \vdash e_1 : S \quad (x : S), \Gamma \vdash e_2 : T}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : T}$$

$$\frac{x : T_1, \Gamma \vdash e : T_2}{\Gamma \vdash \lambda x.e : T_1 \rightarrow T_2} \qquad \frac{\Gamma \vdash e_1 : T_1 \rightarrow T_2 \quad \Gamma \vdash e_2 : T_1}{\Gamma \vdash e_1\, e_2 : T_2}$$

$$\frac{\Gamma, C(X) \vdash e : S \quad X \notin \mathsf{dom}(\Gamma)}{\Gamma \vdash e : \forall X.\, C(X) \Rightarrow S}$$

Type-System

$$\frac{\mathsf{valid}\Gamma \quad x : S \in \Gamma}{\Gamma \vdash x : S}$$

$$\frac{\Gamma \vdash e_1 : S \quad (x : S), \Gamma \vdash e_2 : T}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : T}$$

Old rules:

$$\frac{\mathsf{valid}\ \Gamma \quad (x : S) \in \Gamma \quad S \succ T}{\Gamma \vdash x : T}$$

$$\frac{\Gamma \vdash e_1 : T_1 \quad x : \forall \quad .T_1, \Gamma \vdash e_2 : T_2}{\Gamma \vdash \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 : T_2}$$

$$\frac{x : T_1, \Gamma \vdash e : T_2}{\Gamma \vdash \lambda x.e : T_1 \to T_2} \qquad \frac{\Gamma \vdash e_1 : T_1 \to T_2 \quad \Gamma \vdash e_2 : T_1}{\Gamma \vdash e_1\ e_2 : T_2}$$

Type-System

$$\frac{\mathsf{valid}\,\Gamma \quad x : S \in \Gamma}{\Gamma \vdash x : S}$$

$$\frac{\Gamma \vdash e_1 : S \quad (x : S), \Gamma \vdash e_2 : T}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : T}$$

$$\frac{x : T_1, \Gamma \vdash e : T_2}{\Gamma \vdash \lambda x.e : T_1 \rightarrow T_2} \qquad \frac{\Gamma \vdash e_1 : T_1 \rightarrow T_2 \quad \Gamma \vdash e_2 : T_1}{\Gamma \vdash e_1\, e_2 : T_2}$$

$$\frac{\Gamma, C(X) \vdash e : S \quad X \notin \mathsf{dom}(\Gamma)}{\Gamma \vdash e : \forall X.\, C(X) \Rightarrow S}$$

Type-System

$$\frac{\Gamma \vdash e : \forall X.C(X) \Rightarrow S \quad \Gamma \vdash C(X)[X := T]}{\Gamma \vdash e : S[X := T]}$$

$$\frac{\Gamma \vdash o_1 : S_1 \quad \ldots \quad \Gamma \vdash o_n : S_n}{\Gamma \vdash \{o_1 : S_1, \ldots, o_n : S_n\}}$$

$$\frac{\Gamma \vdash e : S_T \quad \Gamma, o : S_T \vdash p : S'}{\Gamma \vdash \mathsf{inst}\ o : S_T = e\ \mathsf{in}\ p : S'}$$

where we require that Γ contains only a single declaration for every $o : S_T$ (you cannot overload o twice on the same type)

Compilation

- The constraints in $C(X) \Rightarrow T$ represent different implementations for the overloaded function. These constraints are often called dictionaries.
- One can translate the programs with type classes to terms in "standard ML", that is let-polymorphism (one needs to rule out show (read s)).
- However, one can extend the Hindley-Milner algorithm W to deal with type-classes directly.

Research

- We considered only single-parameter type classes. Multi-parameter type classes occur often in practice and are (recently) supported by some Haskell implementations. Multi-parameter need careful design in order to obtain a decidable and meaningful type-system.