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 $\iff$ Formulae
- Typed Terms $\iff$ Proof
- Evaluation $\iff$ Proof Normalisation
- Typing Problem $\iff$ Finding a Proof

We had a look at the Polymorphic Lambda-Calculus - used to encode algebraic datatypes.
“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.
The intuition behind type classes is as follows:

**equal** is a function with type

\[ \text{X} \to \text{X} \to \text{bool} \]

but under the assumption that \( \text{X} \) is of type class **EQ**.
Intuition

The intuition behind type classes is as follows:

- `equal` is a function with type:

\[ \forall X. \ X \to X \to \text{bool} \]

but under the assumption that \( X \) is of type class \( \text{EQ} \).
The intuition behind type classes is as follows

**equal** is a function with type

\[ \forall X. X \rightarrow X \rightarrow \text{bool} \]

but under the assumption that \( X \) is of type class \( \text{EQ} \).

\[ \forall X \text{ such that } X \in \text{EQ}. \; X \rightarrow X \rightarrow \text{bool} \]
The intuition behind type classes is as follows:

- `equal` is a function with type \( \forall X. X \rightarrow X \rightarrow \text{bool} \)
- but under the assumption that \( X \) is of type class \( \text{EQ} \).
- \( \forall X \) such that \( X \in \text{EQ} \). \( X \rightarrow X \rightarrow \text{bool} \)
- \( \forall X. X \in \text{EQ} \Rightarrow X \rightarrow X \rightarrow \text{bool} \)
The intuition behind type classes is as follows:

- `equal` is a function with type\[ \forall X. \ X \rightarrow X \rightarrow \text{bool} \]

but under the assumption that \( X \) is of type class `EQ`.

- \( \forall X \) such that \( X \in \text{EQ} \). \( X \rightarrow X \rightarrow \text{bool} \)
- \( \forall X. \ \text{EQ}(X) \Rightarrow X \rightarrow X \rightarrow \text{bool} \)
The intuition behind type classes is as follows:

- **equal** is a function with type \( \forall X. X \to X \to \text{bool} \)

but under the assumption that \( X \) is of type class **EQ**.

- \( \forall X \text{ such that } X \in \text{EQ}. X \to X \to \text{bool} \)
- \( \forall X. \text{EQ}(X) \Rightarrow X \to X \to \text{bool} \)
- “Types” will be of the form some constraints \( \Rightarrow T \)
Concrete Example

class EQ(\(X\)) where
    equal : \(X \rightarrow X \rightarrow \text{bool}\)

inst equal : int \rightarrow int \rightarrow \text{bool}
    equal = \text{primitive\_equal\_over\_ints}

list\_equal : (equal: \(X \rightarrow X \rightarrow \text{bool}\)) \Rightarrow [X] \rightarrow [X] \rightarrow \text{bool}
    list\_equal [\[] [\[] = \text{True}
    list\_equal (x:xs) (y:ys) = equal x y \land list\_equal xs ys

inst equal : (equal: \(X \rightarrow X \rightarrow \text{bool}\)) \Rightarrow [X] \rightarrow [X] \rightarrow \text{bool}
    equal = \text{list\_equal}
Syntax

Types:

\[ T ::= X \]
\[ T \rightarrow T \]
\[ \text{bool, int, } [X], \ldots \]

Type-schemes:

\[ S ::= T \]
\[ \forall X.C(X) \Rightarrow S \]

Constraints:

\[ C(X) ::= \{ o : X \rightarrow T, \ldots \} \]

where \( T \) can contain \( X \)
Syntax

Terms:

\[ e ::= x \mid e \ e \mid \lambda x. e \mid \text{let } x = e \text{ in } e \]

Programs:

\[ p ::= e \mid \text{inst } o : S_T = e \text{ in } p \]

where \( S \) is type-scheme with the condition that \( T \) can't be a variable
Concrete Syntax

For $\forall X. \; o : X \rightarrow T_1 \Rightarrow T_2$ we write

\[ o : X \rightarrow T_1 \Rightarrow T_2 \]

For instance $o : S = e$ we write

\[ o : S \]
\[ o = e \]

instance equal : (equal: X \rightarrow X \rightarrow bool) \Rightarrow \llbracket X \rrbracket \rightarrow \llbracket X \rrbracket \rightarrow bool

equal = list\_equal
**Type-System**

\[
\text{valid } \Gamma \quad x : S \in \Gamma \quad \frac{}{\Gamma \vdash x : S}
\]

\[
\Gamma \vdash e_1 : S \quad (x : S), \Gamma \vdash e_2 : T \\
\frac{}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : T}
\]

\[
\frac{x : T_1, \Gamma \vdash e : T_2}{\Gamma \vdash \lambda x.e : T_1 \rightarrow T_2}
\]

\[
\frac{\Gamma \vdash e_1 : T_1 \rightarrow T_2}{\Gamma \vdash e_1 \; e_2 : T_2}
\]

\[
\frac{\Gamma, C(X) \vdash e : S \quad X \not\in \text{dom}(\Gamma)}{\Gamma \vdash e : \forall X. C(X) \Rightarrow S}
\]
Type-System

\[
\text{valid} \Gamma \quad x : S \in \Gamma \quad \frac{}{\Gamma \vdash x : S}
\]

\[
\Gamma \vdash e_1 : S \quad (x : S), \Gamma \vdash e_2 : T \quad \frac{}{\Gamma \vdash \text{let} \ x = e_1 \ \text{in} \ e_2 : T}
\]

Old rules:

\[
\text{valid} \quad (x : S) \in \Gamma \quad S \succ T \quad \frac{}{\Gamma \vdash x : T}
\]

\[
\Gamma \vdash e_1 : T_1 \quad x : \forall A.T_1, \Gamma \vdash e_2 : T_2 \quad \frac{}{\Gamma \vdash \text{let} \ x = e_1 \ \text{in} \ e_2 : T_2}
\]

\[
\text{let} \ x = e_1 \ \text{in} \ e_2 : T_2 \quad \frac{x : T_1, \Gamma \vdash e : T_2}{\Gamma \vdash \lambda x.e : T_1 \rightarrow T_2}
\]

\[
\Gamma \vdash e_1 : T_1 \rightarrow T_2 \quad \Gamma \vdash e_2 : T_1 \quad \frac{}{\Gamma \vdash e_1 \ e_2 : T_2}
\]
Type-System

\( \text{valid} \Gamma \quad x : S \in \Gamma \)

\( \Gamma \vdash x : S \)

\( \Gamma \vdash e_1 : S \quad (x : S), \Gamma \vdash e_2 : T \)

\( \Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : T \)

\( x : T_1, \Gamma \vdash e : T_2 \)

\( \Gamma \vdash \lambda x. e : T_1 \rightarrow T_2 \)

\( \Gamma \vdash e_1 : T_1 \rightarrow T_2 \quad \Gamma \vdash e_2 : T_1 \)

\( \Gamma \vdash e_1 \cdot e_2 : T_2 \)

\( \Gamma, C(X) \vdash e : S \quad X \not\in \text{dom}(\Gamma) \)

\( \Gamma \vdash e : \forall X. C(X) \Rightarrow S \)
Type-System

\[ \Gamma \vdash e : \forall X. C(X) \Rightarrow S \quad \Gamma \vdash C(X)[X := T] \]
\[ \Gamma \vdash e : S[X := T] \]
\[ \Gamma \vdash o_1 : S_1 \quad \ldots \quad \Gamma \vdash o_n : S_n \]
\[ \Gamma \vdash \{o_1 : S_1, \ldots, o_n : S_n\} \]
\[ \Gamma \vdash e : S_T \quad \Gamma, o : S_T \vdash p : S' \]
\[ \Gamma \vdash \text{inst } o : S_T = e \text{ in } p : S' \]

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