We extended MinHs to support overloading and type classes:

- \((+) : \text{Num} \Rightarrow a \rightarrow a \rightarrow a\)
- \((==) : \text{Eq} a \Rightarrow a \rightarrow a \rightarrow \text{Bool}\)

New typing rules need the set of predicates \(P\):

\[
\frac{x : \tau \in \Gamma}{P | \Gamma \vdash x : \tau}
\]

\[
\frac{P | \Gamma \vdash e_1 : \tau_1 \quad P | \Gamma \vdash e_2 : \tau_2}{P | \Gamma \vdash \text{apply}(e_1, e_2) : \tau_2}
\]

\[
\frac{P | \Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{P | \Gamma \vdash \text{pair}(e_1, e_2) : (\tau_1 \times \tau_2)}
\]

\[
\frac{P | \Gamma \vdash e : \forall \tau \sigma}{P | \Gamma \vdash e : \forall \tau \sigma}
\]

\[
\frac{P | \Gamma \vdash e : \tau \quad t \notin \text{TV}(\Gamma)}{P | \Gamma \vdash e : \forall \tau}
\]

\[
\frac{\Rightarrow - \text{Elimination}}{P | \Gamma \vdash e : \sigma \Rightarrow \rho \quad P \vdash e}{P | \Gamma \vdash e : \rho}
\]

\[
\frac{\Rightarrow - \text{Introduction}}{P, \pi | \Gamma \vdash e : \rho \quad P | \Gamma \vdash e : \pi \Rightarrow \rho}{P | \Gamma \vdash e : \sigma \Rightarrow \rho}
\]

**Overloading Resolved**

At some point, overloaded functions have to be instantiated to the concrete operation:

- \(1.0 + 1.5 \leadsto 1.0 + \text{Float} 1.5\)
- \((1, \text{True}) == (1, \text{False}) \leadsto (1 ==_{\text{Int}} 1) \& \& (\text{True} ==_{\text{Bool}} \text{False})\)

**In OO-languages:**

- Objects “know” which method to apply
- Java, C++ objects have pointers to a vtable, which contains all the methods of the class
- Would not work for MinHs, since elements of a data type cannot know which operations might be applied to them
Idea:

- Table of the argument class is passed to overloaded function
- The table is called a dictionary
- Overloaded function picks the right concrete function from dictionary
- Dictionary (simplified) for Eq Int is a pair of the functions == on Int and /= on Int:
  - (==_{int} /=_{int})
- The overloaded function == is a function which takes a dictionary as argument (always a pair for class Eq), and returns the first component (second component for /=)

This means that == internally has type

- EqDict a \rightarrow a \rightarrow a \rightarrow Bool where EqDict a stands for the type of the dictionary of Eq a instead of
- Eq a \Rightarrow a \rightarrow a \rightarrow Bool

We call the type of the dictionary for class C and instance a simply C a, so the internal type of == is

- Eq a \rightarrow a \rightarrow a \rightarrow Bool

**TYPE INFERENCE ALGORITHM**

The type inference algorithm serves now three tasks

- derive type of expression
- translate expressions into internal representation (add dictionaries)
- collect constraints in the sequence P (as we will see, the order of the constraints is now important)

**THE ROLE OF P**

- So far, P was simply a set containing all constraints
- For the type inference algorithm, the role and format of P is different
- P consists of two n-tuples of the form
  
- (d_1, \ldots, d_n) : (\pi_1, \ldots, \pi_n)

  where the d_i are variable names referring to dictionaries and \pi_i constraints. The variable d_1 is bound to the dictionary of \pi_1, and so on
- For example P = (d_1, d_2) : (Eq Int, Num Float)
- So, d_i refers to the dictionary Eq Int, d_2 to Num Float.
- We write P P' to denote the combination of constraint sequences
\[
x : \forall a_1 \ldots \forall a_n. \pi_1 \Rightarrow \ldots \pi_n \Rightarrow \tau \in \Gamma \quad \text{and } d_i \text{ fresh}
\]
\[
(d_1, \ldots, d_n) : [a_i/a_j](\pi_1, \ldots, \pi_n) \mid \Gamma \vdash x \rightsquigarrow \text{apply}(x, (d_1, \ldots, d_n) : [a_i/a_j] \tau).
\]

\[
P \mid TT \vdash e_1 \rightsquigarrow e_1' : \tau_1 \quad P' \mid TTT \vdash e_2 \rightsquigarrow e_2' : \tau_2 \quad T' \tau_1 \overset{U}{\Rightarrow} \tau_2 \overset{\alpha}{\Rightarrow} \alpha 
\]

\[
\frac{U(TP, P') \mid UTT \vdash \text{apply}(e_1, e_2) \rightsquigarrow \text{apply}(e_1', e_2') : U\alpha}{\text{fresh}}
\]

**Slide 9**

\[
P \mid T(\Gamma \cup \{x : \alpha_1\} \cup \{f : \alpha_2\}) \vdash e \rightsquigarrow e' : \tau \quad \alpha_2 \overset{U}{\Rightarrow} \alpha \overset{\tau}{\Rightarrow} \tau
\]

\[
\frac{U^{P} \mid UT \vdash \text{letfun}(f.x.e) \rightsquigarrow \text{letfun}(f.x.e') : U\alpha_2}{\text{fresh}}
\]

\[
d : P \mid TT \vdash e_1 \rightsquigarrow e_1' : \tau \quad TTT \vdash (x : \text{Gen}(\Gamma, \tau)) \vdash e_2 \rightsquigarrow e_2' : \tau'
\]

\[
\frac{TTT \vdash \text{let}(e_1, x.e_2) \rightsquigarrow \text{let}(\text{letfun}(f.d_1', x.e_2') : \tau')}{\text{fresh}}
\]

The Role of \( P \)}