We extended MinHs to support overloading and type classes:

\[\rightarrow (\text{+}) :: \text{Num} \ a \Rightarrow a \rightarrow a \rightarrow a\]

\[\rightarrow (==) :: \text{Eq} \ a \Rightarrow a \rightarrow a \rightarrow \text{Bool}\]

New typing rules need the set of predicates $P$:

\[
\begin{align*}
\frac{x : \tau \in \Gamma}{P | \Gamma \vdash x : \tau} & \quad \frac{P | \Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad P | \Gamma \vdash e_2 : \tau_1}{P | \Gamma \vdash \text{apply}(e_1, e_2) : \tau_2} \\
\frac{P | \Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{P | \Gamma \vdash \text{pair}(e_1, e_2) : (\tau_1 \times \tau_2)} \\
\frac{P | \Gamma \vdash e : \forall t.\tau}{P | \Gamma \vdash e : [\tau_1/t]\tau} \\
\frac{P | \Gamma \vdash e : \tau \quad t \notin \text{TV}(\Gamma)}{P | \Gamma \vdash e : \forall t.\tau}
\end{align*}
\]
\( \Rightarrow \)  \( \Rightarrow \)- Elimination:

\[
\frac{P \mid \Gamma \vdash e : \pi \Rightarrow \rho \quad P \vdash \pi}{P \mid \Gamma \vdash e : \rho}
\]

\( \Rightarrow \)  \( \Rightarrow \)-Introduction:

\[
\frac{P, \pi \mid \Gamma \vdash e : \rho}{P \mid \Gamma \vdash e : \pi \Rightarrow \rho}
\]
At some point, overloaded functions have to be instantiated to the concrete operation:

- \( 1.0 + 1.5 \mapsto 1.0 +_{\text{float}} 1.5 \)
- \((1, \text{True}) == (1, \text{False}) \mapsto \)
  \((1 ==_{\text{Int}} 1) \&\& (\text{True} ==_{\text{Bool}} \text{False})\)
In OO-languages:

- Objects “know” which method to apply
- Java, C++ objects have pointer to a vtable, which contains all the methods of the class
In OO-languages:

- Objects “know” which method to apply
- Java, C++ objects have pointer 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`:
  - `==(\text{Int}, \neq_{\text{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

\[ \text{EqDict } a \rightarrow a \rightarrow a \rightarrow \text{Bool} \]

where `EqDict a` stands for the type of the dictionary of `Eq a` instead of

\[ \text{Eq } a \Rightarrow a \rightarrow a \rightarrow \text{Bool} \]
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

- $Eq \ a \rightarrow \ a \rightarrow \ a \rightarrow \ Bool$
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
  
  $\begin{align*}
  & (d_1, \ldots, d_n) : (\pi_1, \ldots, \pi_n) \\
  \end{align*}$

  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, $d_1$ refers to the dictionary $Eq\ Int$, $d_2$ to $Num\ Float$.

→ We write $PP'$ to denote the combination of constraint sequences
\[ x : \forall a_1 \ldots \forall a_n. \pi_1 \Rightarrow \ldots \pi_n \Rightarrow \tau \in \Gamma \quad \beta_i \text{ and } d_i \text{ fresh} \]

\[(d_1, \ldots, d_n) : [\beta_i/a_i](\pi_1, \ldots, \pi_n) \mid \Gamma \vdash x \rightsquigarrow \text{apply}(x, (d_1, \ldots, d_n) : [\beta_i/a_i] \tau, \text{fresh}) \]

\[
P \mid T\Gamma \vdash e_1 \rightsquigarrow e'_1 : \tau_1 \quad P' \mid T'T\Gamma \vdash e_2 \rightsquigarrow e'_2 : \tau_2 \quad T'\tau_1 \overset{U}{\sim} \tau_2 \to \alpha \quad \alpha \text{ fresh} \]

\[
U(T'P, P') \mid UT'T\Gamma \vdash \text{apply}(e_1, e_2) \rightsquigarrow \text{apply}(e'_1, e'_2) : U\alpha \]

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

\[
UP \mid UT\Gamma \vdash \text{letfun}(f.x.e) \rightsquigarrow \text{letfun}(f.x.e') : U\alpha_2 \]

\[
d : P \mid T\Gamma \vdash e_1 \rightsquigarrow e'_1 : \tau \quad T'T\Gamma \cup \{x : \text{Gen}(\Gamma, \tau)\} \vdash e_2 \rightsquigarrow e'_2 : \tau' \]

\[
T'T\Gamma \vdash \text{let}(e_1, x.e_2) \rightsquigarrow \text{let}(\text{letfun}(f.d.e'_1), x.e'_2) : \tau' \]