Polymorphic Types (cont)
Review

- Impredicative variant of System F very expressive:
  - can express complex polymorphic functions
  - type abstraction and application
  - limited form of self-application
  - prove interesting theorems based on type structure
  - complicated semantics
    - termination proof
    - type reconstruction
Predicative Polymorphism

- Restriction: type variables can be instantiated only with monomorphic types
- This restriction can be expressed syntactically
  \[ \tau ::= b \mid \tau_1 \to \tau_2 \mid t \]
  \[ \sigma ::= \tau \mid \forall t. \sigma \mid \sigma_1 \to \sigma_2 \]
  \[ e ::= x \mid e_1 \, e_2 \mid \lambda x: \sigma. \, e \mid \Lambda t. e \mid e[\tau] \]

  - Type application is restricted to mono types
  - Cannot apply “id” to itself anymore

- Same typing rules
- Simple semantics and termination proof
- Type reconstruction still undecidable
- Must restrict further!
Prenex Predicative Polymorphism

- Restriction: polymorphic type constructor at top level only
- This restriction can also be expressed syntactically
  \[ \tau ::= b \mid \tau_1 \rightarrow \tau_2 \mid \top \]
  \[ \sigma ::= \tau \mid \forall \tau. \sigma \]
  \[ e ::= x \mid e_1 \ e_2 \mid \lambda x: \tau. \ e \mid \Lambda \tau. \ e \mid e [\tau] \]
- Type application is restricted to mono types (i.e., predicative)
- Abstraction only on mono types
- The only occurrences of \( \forall \) are at the top level of a type
  \( (\forall \tau. \top \rightarrow \top) \rightarrow (\forall \tau. \top \rightarrow \top) \) is not a valid type

- Same typing rules
- Simple semantics and termination proof
- Decidable type inference!
Expressiveness of Prenex Predicative $F_2$

- We have simplified too much!

- Not expressive enough to encode nat, bool
  - bool = $\forall t. t \rightarrow t \rightarrow t$
  - true = $\forall t. \lambda x:t.\lambda y:t. x$
  - false = $\forall t. \lambda x:t.\lambda y:t. y$
  - But such encodings are only of theoretical interest anyway

- Is it expressive enough in practice?
  - Almost
  - Cannot write something like
    $$(\lambda s:\forall t.\tau. \ldots s \ [nat] \ x \ldots \ s \ [bool] \ y) (\forall t. \ldots \ code\ for\ sort)$$
  - Because the type of formal argument $s$ cannot be polymorphic
ML’s Polymorphic Let

- **ML solution:** slight extension of the predicative $F_2$
  - Introduce “let $x : \sigma = e_1$ in $e_2$”
  - With the semantics of “$(\lambda x : \sigma. e_2) \, e_1$”
  - And typed as “[e_1/x] e_2”

\[
\Gamma \vdash e_1 : \sigma \quad \Gamma, x : \sigma \vdash e_2 : \tau \\
\Gamma \vdash \text{let } x : \sigma = e_1 \text{ in } e_2 : \tau
\]

- This lets us write the polymorphic sort as
  
  ```
  let 
  s : \forall t. \tau = \Lambda t. ... \text{ code for polymorphic sort } ... 
  in 
  ... s \, [\text{nat}] \times .... s \, [\text{bool}] \, y
  ```

- **Surprise:** this was a major ML design flaw!
ML Polymorphism and References

- let is evaluated using call-by-value but is typed using call-by-name
  - What if there are side effects?
- Example:
  
  ```ml
  let x : ∀t. (t ! t) ref = ∀t.ref (λx : t. x)
  in
  x [bool] := λx: bool. not x
  (! x [int]) 5
  end
  ```
  - Will apply “not” to 5
  - Similar examples can be constructed with exceptions
- It took 10 years to find and agree on a clean solution
The Value Restriction in ML

- A type in a let is generalized only for syntactic values

\[
\frac{\Gamma \vdash e_1 : \sigma \quad \Gamma, x : \sigma \vdash e_2 : \tau}{\Gamma \vdash \text{let } x : \sigma = e_1 \text{ in } e_2 : \tau}
\]

- \(e_1\) is a syntactic value or \(\sigma\) is monomorphic.

- Since \(e_1\) is a value, its evaluation cannot have side-effects.

- In this case call-by-name and call-by-value are the same.

- In the previous example \(\text{ref } (\lambda x : t. x)\) is not a value.

- This is not too restrictive in practice!