The Limitations of F₁ (simply-typed λ-calculus)

- In F₁ each function works exactly for one type
- Example: the identity function
  - \( \text{id} = \lambda x : \tau. \ x : \tau \rightarrow \tau \)
  - We need to write one version for each type
  - Even more important: \( \text{sort} : (\tau \rightarrow \tau \rightarrow \text{bool}) \rightarrow \tau \text{ array} \rightarrow \text{unit} \)
- The various sorting functions differ only in typing
  - At runtime they perform exactly the same operations
  - We need different versions only to keep the type checker happy
- Two alternatives:
  - Circumvent the type system (see C, Java, ...), or
  - Use a more flexible type system that lets us write only one sorting function
Polymorphism

- Informal definition
  A function is polymorphic if it can be applied to “many” types of arguments

- Various kinds of polymorphism depending on the definition of “many”
  - subtype (or bounded) polymorphism
    “many” = all subtypes of a given type
  - ad-hoc polymorphism
    “many” = depends on the function
    choose behavior at runtime (depending on types, e.g. sizeof)
  - parametric predicative polymorphism
    “many” = all monomorphic types
  - parametric impredicative polymorphism
    “many” = all types
Parametric Polymorphism: Types as Parameters (System F)

- We introduce type variables and allow expressions to have variable types
- We introduce polymorphic types
  \[ \tau ::= b \mid \tau_1 \rightarrow \tau_2 \mid t \mid \forall t. \tau \]

  \[ e ::= x \mid \lambda x : \tau . e \mid e_1 e_2 \mid \forall t. e \mid e[\tau] \]

  - \[ \forall t. e \] is type abstraction (or generalization)
  - \[ e[\tau] \] is type application (or instantiation)

- Examples:
  - \[ id = \forall t. \lambda x : t . x \quad : \quad \forall t. t \rightarrow t \]
  - \[ id[int] = \lambda x : \text{int} . x \quad : \quad \text{int} \rightarrow \text{int} \]
  - \[ id[bool] = \lambda x : \text{bool} . x \quad : \quad \text{bool} \rightarrow \text{bool} \]
  - “id 5” is invalid. Use “id [int] 5” instead
Impredicative Polymorphism

- The typing rules:

\[
\begin{align*}
\frac{x : \tau \text{ in } \Gamma}{\Gamma \vdash x : \tau} & \quad \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x : \tau. e : \tau \to \tau'} \\
\frac{\Gamma \vdash e_1 : \tau \to \tau' \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 \; e_2 : \tau'} \\
\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \forall t. e : \forall t. \tau} & \quad \text{\( t \) does not occur in } \Gamma \\
\frac{\Gamma \vdash e : \forall t. \tau'}{\Gamma \vdash e[\tau] : [\tau / t] \tau'}
\end{align*}
\]
Impredicative Polymorphism (Cont.)

- Verify that “id [int] 5” has type int
- Note the side-condition in the rule for type abstraction
  - Prevents ill-formed terms like: \( \lambda x : t. \Lambda t. x \)
- The evaluation rules are just like those of \( F_1 \)
  - This means that type abstraction and application are all performed at compile time
  - We do not evaluate under \( \Lambda \) (\( \Lambda t. e \) is a value)
  - We do not have to operate on types at run-time
  - This is called phase separation: type checking and execution
Observations

- Based on the type of a term we can prove properties of that term
- There is only one value of type $\forall t.t \to t$
  - The polymorphic identity function
- There is no value of type $\forall t.t$
- Take the function: $\text{reverse} : \forall t. \; t \text{ List} \to t \text{ List}$
  - This function cannot inspect the elements of the list
  - It can only produce a permutation of the original list
  - If $L_1$ and $L_2$ have the same length and let “match” be a function that compares two lists element-wise according to an arbitrary predicate
  - then “match $L_1 L_2$” $\equiv$ “match (reverse $L_1$) (reverse $L_2$)” !
Expressiveness of Impredicative Polymorphism

- This calculus is called
  - $F_2$
  - system F
  - second-order $\lambda$-calculus
  - polymorphic $\lambda$-calculus
- Polymorphism is extremely expressive
- We can encode many base and structured types in $F_2$
Encoding Base Types in F₂

- **Booleans**
  - \( \text{bool} = \forall t. t \to t \to t \) (given any two things, select one)
  - There are exactly two values of this type!
    - \( \text{true} = \Lambda t. \lambda x:t. \lambda y:t. x \)
    - \( \text{false} = \Lambda t. \lambda x:t. \lambda y:t. y \)
    - \( \text{not} = \lambda b: \text{bool}. \Lambda t. \lambda x:t. \lambda y:t. b [t] y x \)

- **Naturals**
  - \( \text{nat} = \forall t. (t \to t) \to t \to t \) (given a successor and a zero element, compute a natural number)
    - \( 0 = \Lambda t. \lambda s:t \to t. \lambda z:t. z \)
    - \( n = \Lambda t. \lambda s:t \to t. \lambda z:t. s (s...s(n)) \)
    - \( \text{add} = \lambda n: \text{nat}. \lambda m: \text{nat}. \Lambda t. \lambda s:t \to t. \lambda z:t. n [t] s (m [t] s z) \)
    - \( \text{mul} = \lambda n: \text{nat}. \lambda m: \text{nat}. \Lambda t. \lambda s:t \to t. \lambda z:t. n [t] (m [t] s) z \)
Expressiveness of \( F_2 \)

- **Polymorphic application**
  - \( \text{selfApp} = \)
    \[
    \lambda x : \forall t. t \to t . \lambda y : t. x [\forall t. t \to t] x : (\forall t. t \to t) \to (\forall t. t \to t)
    \]
  - \( \text{double} = \Lambda t. \lambda f : t \to t . \lambda a : t . f(f(a)) : (\forall t. t \to t) \to t \to t \)
  - \( \text{quadruple} = \)
    \[
    \Lambda t. \text{double}[t \to t] (\text{double}[t]) : (\forall t. t \to t) \to t \to t
    \]

- **Recursive types?**
  - We can encode primitive recursion but not full recursion
Assessment

- 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
  \[ \tau ::= b \mid \tau_1 \to \tau_2 \mid \top \]
  \[ \sigma ::= \tau \mid \forall \cdot \sigma \mid \sigma_1 \to \sigma_2 \]
  \[ e ::= x \mid e_1 e_2 \mid \lambda x: \sigma. \ e \mid \Lambda t. e \mid e [\tau] \]
  - 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
  \[ \tau ::= b \mid \tau_1 \rightarrow \tau_2 \mid t \]
  \[ \sigma ::= \tau \mid \forall \tau. \sigma \]
  \[ e ::= x \mid e_1 \; e_2 \mid \lambda x: \tau. \; e \mid \Lambda t. \; e \mid e [\tau] \]
  - 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
    \[ (\forall \tau. \; t \rightarrow t) \rightarrow (\forall \tau. \; t \rightarrow t) \] is not a valid type
- Same typing rules
- Simple semantics and termination proof
- Decidable type inference!