The Limitations of $F_1$ (simply-typed $\lambda$-calculus)

- In $F_1$ each function works exactly for one type
- Example: the identity function
  - $id = \lambda x: \tau. \; x : \tau \rightarrow \tau$
  - We need to write one version for each type
  - Even more important: $sort : (\tau \rightarrow \tau \rightarrow \text{bool}) \rightarrow \tau \text{ array} \rightarrow \text{unit}$
- 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:
  - \( \text{id} = \forall t. \lambda x: t. x \quad : \quad \forall t. t \rightarrow t \)
  - \( \text{id[int]} = \lambda x: \text{int}. x \quad : \quad \text{int} \rightarrow \text{int} \)
  - \( \text{id[bool]} = \lambda x: \text{bool}. x \quad : \quad \text{bool} \rightarrow \text{bool} \)
  - “id 5” is invalid. Use “id [int] 5” instead
Impredicative Polymorphism

- The typing rules:

\[ \Gamma \vdash x : \tau \]

\[ \Gamma, x : \tau \vdash e : \tau' \]

\[ \Gamma \vdash \lambda x : \tau. e : \tau \rightarrow \tau' \]

\[ \begin{array}{c}
\Gamma \vdash e_1 : \tau \rightarrow \tau' \\
\Gamma \vdash e_2 : \tau
\end{array} \quad \Gamma \vdash e_1 \ e_2 : \tau' \]

\[ \Gamma \vdash e : \tau \]

\[ \Gamma \vdash \Lambda t. e : \forall t. \tau \]  \quad \text{\textit{t does not occur in } } \Gamma

\[ \Gamma \vdash e : \forall t. \tau' \]

\[ \Gamma \vdash e[\tau] : [\tau/t] \tau' \]
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 F₁
  - 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 $\forall t. t \rightarrow t$
  - The polymorphic identity function
- There is no value of type $\forall t. t$
- Take the function: reverse : $\forall t. t \text{ List} \rightarrow t \text{ List}$
  - 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**
  - bool = ∀t.t → t → t (given any two things, select one)
  - There are exactly two values of this type!
    - true = Λt. λx:t.λy:t. x
    - false = Λt. λx:t.λy:t. y
  - not = λb:bool. Λt.λx:t.λy:t. b [t] y x

- **Naturals**
  - nat = ∀t. (t → t) → t → t (given a successor and a zero element, compute a natural number)
  - 0 = Λt. λs:t → t.λz:t. z
  - n = Λt. λs:t → t.λz:t. s (s (s...s(n)))
  - add = λn:nat. λm:nat. Λt. λs:t → t.λz:t. n [t] s (m [t] s z)
  - mul = λn:nat. λm:nat. Λt. λs:t → t.λz:t. n [t] (m [t] s) z
Expressiveness of $F_2$

- We can encode similarly:
  - $\tau_1 + \tau_2$ as $\forall t. (\tau_1 \rightarrow t) \rightarrow (\tau_2 \rightarrow t) \rightarrow t$
  - $\tau_1 \times \tau_2$ as $\forall t. (\tau_1 \rightarrow \tau_2 \rightarrow t) \rightarrow t$
  - $\text{unit}$ as $\forall t. t \rightarrow t$

- Polymorphic application:
  - $\text{selfApp} = \lambda x: \forall t. t \rightarrow t. x[\forall t. t \rightarrow t] x : (\forall t. t \rightarrow t) \rightarrow (\forall t. t \rightarrow t)$
  - $\text{double} = \forall t. \lambda f : t \rightarrow t. \lambda a : t. f(f(a)) : \forall t. (t \rightarrow t) \rightarrow t \rightarrow t$
  - $\text{quadruple} = \forall t. \text{double} [t \rightarrow t] (\text{double} [t]) : \forall t. (t \rightarrow t) \rightarrow t \rightarrow t$

- We cannot encode $\mu t. t$
  - We can encode primitive recursion but not full recursion
  - All terms in $F_2$ have a termination proof in second-order Peano arithmetic (Girard, 1971): strongly normalizing
    - This is the set of naturals defined using zero, successor, induction along with quantification both over naturals and over sets of naturals