Extending the (Simply) Typed $\lambda$-Calculus to $\lambda$-Languages with Constants
Accommodating Standard Ground Types

Any realistic statically typed function language includes ground types like `bool` and `int` (and many more such as `char` and `float`).

Hence, the definition of types (type expressions) \( \tau \) looks like:

\[
\tau ::= \text{int} \mid \text{bool} \mid \ldots \mid \tau \to \tau
\]

and the base environment (which is empty in the simply typed lambda calculus) contains types for all of the primitive functions and operators. (We interpret operators as abbreviated syntax for conventional function applications.)

To support multi-ary functions, we also need to introduce Cartesian products of types \( \tau \times \tau \) as an extra clause after \( \tau \to \tau \) above. We will use this extension without further comment when we need it.
How Do We Type Functional Constructs?

Examples:

- Conditional expressions
- New algebraic (inductively defined) data types
- Recursive let

General answer:

(i) for each new syntactic construct like conditionals, we introduce a new rule;

(ii) for each new form of data (which only requires adding new constants [including functions] to the language), we simply augment the set of type constructors by the new type constructor (e.g. int-list) and augment the contents of the base type environment by the new constants and their types.

Let's look at each example in our list above.
Typing Conditional Expressions

Note: if conditional expressions are simply written as applications of a ternary \texttt{if} operator, then all we need to do is add \texttt{if} to the set of constant symbols and add the type of \texttt{if} to the base type environment (assuming our typing framework is polymorphic which we will explain later). In fact, our new rule for the \texttt{if} construct simply codifies the same typing constraints (but does not introduce a new constant symbol or rely on polymorphism).

Our conditional expressions presume the existence of a bool type

\[
\Gamma \vdash B : \text{bool}; \quad \Gamma \vdash M : \tau; \quad \Gamma \vdash N : \tau
\]

\[
\frac{\Gamma \vdash \text{if } B \text{ then } M \text{ else } N : \tau}{\Gamma \vdash \text{if } B \text{ then } M \text{ else } N : \tau}
\]
Assume we define some new form of data in a program. In structurally typed programming languages, all data values have a unique type. Hence, when a new union type is introduced, all values of that type must be disjoint from all existing types. This invariant is maintained by forcing all of the components of a new union type to be tagged. This data construction is called a discriminated union.

Example: binary trees

\[
BT ::= \text{leaf}(\text{int}) \mid \text{make-BT}(BT, BT)
\]

In ML-like languages, new forms of data are introduced in `datatype` declarations which have an abstract syntax similar to our example. In ML, the names of accessors (selectors) are implicit because pattern-matching notation is used to extract the fields of constructed data objects.
Typing New Forms of Data cont.

Given

\[ \text{BT} :: = \text{make-leaf}(\text{int}) \mid \text{make-BT}(\text{BT}, \text{BT}) \]

we augment the set of type expressions by the primitive type \text{BT} and the base typing environment by the declarations:

- \text{make-leaf}: \text{int} \rightarrow \text{BT}
- \text{make-BT}: \text{BT} \times \text{BT} \rightarrow \text{BT}
- \text{leaf-1}: \text{BT} \rightarrow \text{int}
- \text{BT-1}: \text{BT} \rightarrow \text{BT}
- \text{BT-2}: \text{BT} \rightarrow \text{BT}

assuming that we use the name \text{leaf-1} for the accessor for \text{make-leaf} and the names \text{BT-1} and \text{BT-2} for the accessors for \text{make-BT}. In languages with pattern matching like ML, the names of the accessors (can be implicit because pattern matching notation makes them unnecessary.

In ML-like languages, data type definitions are typically lexically scoped so \texttt{datatype} statements have an abstract syntax like \texttt{let} with an explicit body which is the scope of the definition. We will ignore scoping for data definitions and fix the data types for any program that we consider. (In Java, data definitions do not have scope. Visibility is an administrative notion not a semantic one.)

**Key intuition:** to accommodate new forms of data, we only need to add new primitive types (corresponding to the new data) and entries (the types of new constants) to the base type environment \( \Gamma_0 \).
Typing Let and Recursive Let

The typing rule for pure `let` is simply an abbreviation for appropriately combining the abstraction and application rules:

\[
\frac{\Gamma \vdash M : \sigma; \Gamma \cup \{x : \sigma\} \vdash N : \tau}{\Gamma \vdash \text{let } x : \sigma := M \text{ in } N : \tau}
\]  
(pure let rule)

The corresponding rule for recursive `let` is:

\[
\frac{\Gamma \cup \{x : \sigma\} \vdash M : \sigma; \Gamma \cup \{x : \sigma\} \vdash N : \tau}{\Gamma \vdash \text{let } x : \sigma := M \text{ in } N : \tau}
\]  
(letrec rule)

which differs only in one small (but very important!) detail: the type environment for proving `M : \sigma`.
Type Reconstruction

• Given an untyped expression $M$ in a $\lambda$-language with constants, can we determine if it has a typing in the corresponding simply typed $\lambda$-language?

• Yes! Moreover the reconstructed type $\tau$ (sometimes ambiguously called the inferred type) may contain free type variables that can be instantiated (replaced) by arbitrary type expressions (perhaps containing type variables. These instantiations correspond to valid types for $\tau$. The reconstructed type $\tau$ with the free type variables is called the principal (most general) type of $M$. Languages in the ML family (Haskell, Ocaml, …) perform type reconstruction.