Overview

- So far, we've covered
  - arithmetic
  - booleans, conditionals (if then else)
  - variables and simple binding (let)
- Let allows us to compute values of expressions
- and use variables to store intermediate values
- but not to define *computations* on unknown values.
- That is, there is no feature analogous to Haskell’s functions, Scala’s `def`, or methods in Java.
- Today, we consider *functions* and *recursion*

Named functions

A simple way to add support for functions is as follows:

\[ e ::= \cdots \mid f(e) \mid \text{let fun } f(x:\tau) = e_1 \text{ in } e_2 \]

- Meaning: Define a function called \( f \) that takes an argument \( x \) and whose result is the expression \( e_1 \).
- Make \( f \) available for use in \( e_2 \).
- (That is, the scope of \( x \) is \( e_1 \), and the scope of \( f \) is \( e_2 \).)
- This is pretty limited:
  - for now, we consider one-argument functions only.
  - no recursion
  - functions are not first-class “values” (e.g. can’t pass a function as an argument to another)

Examples

- We can define a squaring function:

  \[
  \text{let fun } \text{square}(x:\text{int}) = x \times x \text{ in } \cdots
  \]

- or (assuming inequality tests) absolute value:

  \[
  \text{let fun } \text{abs}(x:\text{int}) = \text{if } x < 0 \text{ then } -x \text{ else } x \text{ in } \cdots
  \]
Types for named functions

- We introduce a type constructor \( \tau_1 \rightarrow \tau_2 \), meaning “the type of functions taking arguments in \( \tau_1 \) and returning \( \tau_2 \).
- We can typecheck named functions as follows:
  \[
  \Gamma, x : \tau_1 \vdash e_1 : \tau_2 \quad \Gamma, f : \tau_1 \rightarrow \tau_2 \vdash e_2 : \tau
  \]
  \[
  \Gamma \vdash \text{let fun } f(x : \tau_1) = e_1 \text{ in } e_2 : \tau
  \]
  \[
  \Gamma(f) = \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e : \tau_1
  \]
  \[
  \Gamma \vdash f(e) : \tau_2
  \]
- For convenience, we just use a single environment \( \Gamma \) for both variables and function names.

Example

Typechecking of \( \text{abs}(-42) \)

\[
\begin{align*}
\Gamma(x) &= \text{int} & \Gamma(x) &= \text{int} & \Gamma(x) &= \text{int} \\
\Gamma \vdash x : \text{int} & \quad \Gamma \vdash 0 : \text{int} & \quad \Gamma \vdash x : \text{int} & \quad \Gamma \vdash -x : \text{int} & \quad \Gamma \vdash x : \text{int} \\
\Gamma \vdash x < 0 : \text{bool} & \quad \Gamma \vdash \text{if } x < 0 \text{ then } -x \text{ else } x : \text{int} \\
\Gamma \vdash e_{\text{abs}} : \text{int} & \quad \Gamma \vdash \text{abs} : \text{int} \rightarrow \text{int} & \quad \Gamma \vdash -42 : \text{int} \\
\Gamma \vdash \text{let fun } \text{abs}(x : \text{int}) = e_{\text{abs}} \text{ in } \text{abs}(-42) : \text{int} & \quad \text{where } e_{\text{abs}} = \text{if } x < 0 \text{ then } -x \text{ else } x \text{ and } \Gamma = x : \text{int}. 
\end{align*}
\]

Semantics of named functions

- We can define rules for evaluating named functions as follows.
- First, let \( \delta \) be an environment mapping function names \( f \) to their “definitions”, which we’ll write as \( \langle x \Rightarrow e \rangle \).
- When we encounter a function definition, add it to \( \delta \).
  \[
  \delta[f \mapsto \langle x \Rightarrow e \rangle], e_2 \Downarrow \nu
  \]
  \[
  \delta, \text{let fun } f(x : \tau) = e_1 \text{ in } e_2 \Downarrow \nu
  \]
- When we encounter an application, look up the definition and evaluate the body with the argument value substituted for the argument:
  \[
  \delta, e_0 \Downarrow \nu \quad \delta(f) = \langle x \Rightarrow e \rangle \quad \delta, e_0[v_0/x] \Downarrow \nu
  \]
  \[
  \delta, f(e_0) \Downarrow \nu
  \]

Example

Evaluation of \( \text{abs}(-42) \)

\[
\begin{align*}
\delta, -42 < 0 \Downarrow \text{true} & \quad \delta, -(42) \Downarrow 42 \\
\delta, \text{if } -42 < 0 \text{ then } -(42) \text{ else } -42 \Downarrow 42 \\
\delta, -42 \Downarrow -42 & \quad \delta(\text{abs}) = \langle x \Rightarrow e_{\text{abs}} \rangle \quad \delta, e_{\text{abs}}[-42/x] \Downarrow 42 \\
\delta, \text{abs}(-42) \Downarrow 42 & \quad \text{let fun } \text{abs}(x : \text{int}) = e_{\text{abs}} \text{ in } \text{abs}(-42) \Downarrow 42 \\
\end{align*}
\]

where \( e_{\text{abs}} = \text{if } x < 0 \text{ then } -x \text{ else } x \) and
\[
\delta = [\text{abs} \mapsto \langle x \Rightarrow e_{\text{abs}} \rangle]
\]
### Static vs. Dynamic Scope

- The terms *static* and *dynamic* scope are sometimes used.
- In **static scope**, the scope and binding occurrences of all variables can be determined from the program text, **without** actually running the program.
- In **dynamic scope**, this is not necessarily the case: the scope of a variable can depend on the context in which it is evaluated **at run time**.

#### Function Bodies Can Contain Free Variables

Consider:

```plaintext
let x = 1 in
let fun f(y : int) = x + y in
let x = 10 in f(3)
```

Here, `x` is bound to 1 at the time `f` is defined, but re-bound to 10 when by the time `f` is called.

- There are two reasonable-seeming result values, depending on which `x` is **in scope**:
  - **Static scope** uses the binding `x = 1` present when `f` is defined, so we get `1 + 3 = 4`.
  - **Dynamic scope** uses the binding `x = 10` present when `f` is used, so we get `10 + 3 = 13`.

### Dynamic Scope Breaks Type Soundness

- Even worse, what if we do this:
  ```plaintext
  let x = 1 in
  let fun f(y : int) = x + y in
  let x = true in f(3)
  ```
  When we typecheck `f`, `x` is an integer, but it is re-bound to a boolean by the time `f` is called.
  - The program as a whole typechecks, but we get a run-time error: **dynamic scope makes the type system unsound!**
  - Early versions of LISP used dynamic scope, and it is arguably useful in an untyped language.
  - Dynamic scope is now generally acknowledged as a mistake — but one that naive language designers still make.

### Anonymous, First-Class Functions

- In many languages (including Java as of version 8), we can also write an expression for a function without a name:
  ```plaintext
  \lambda x : \tau. e
  ```
- Here, `\lambda` (Greek letter lambda) introduces an anonymous function expression in which `x` is bound in `e`.
  - (The `\lambda`-notation dates to Church’s higher-order logic (1940); there are several competing stories about why he chose `\lambda`.)
- In Scala one writes: `(x : Type) => e`
- In Java 8: `x -> e` (no type needed)
- In Haskell: `\x -> e` or `\x :: Type -> e`
- The **lambda-calculus** is a model of anonymous functions
Types for the λ-calculus

- We define \( L_{\text{Lam}} \) to be \( L_{\text{Let}} \) extended with typed λ-abstraction and application as follows:

  \[
  e ::= \cdots | e_1 e_2 | \lambda x: \tau. \ e \\
  \tau ::= \cdots | \tau_1 \rightarrow \tau_2
  \]

- \( \tau_1 \rightarrow \tau_2 \) is (again) the type of functions from \( \tau_1 \) to \( \tau_2 \).

- We can extend the typing rules as follows:

  \[
  \Gamma \vdash e : \tau \quad \text{for } L_{\text{Lam}}
  \]

  \[
  \begin{array}{c}
  \Gamma, x: \tau_1 \vdash e : \tau_2 \\
  \Gamma \vdash \lambda x: \tau_1. e : \tau_1 \rightarrow \tau_2 \\
  \Gamma \vdash e_1 : \tau_1 \\
  \Gamma \vdash e_2 : \tau_2 \\
  \Gamma \vdash e_1 e_2 : \tau_2
  \end{array}
  \]

Examples

- In \( L_{\text{Lam}} \), we can define a higher-order function that calls its argument twice:

  \[
  \text{let fun } \text{twice}(f : \tau \rightarrow \tau) = \lambda x: \tau. f(f(x)) \text{ in } \cdots
  \]

- and we can define the composition of two functions:

  \[
  \text{let compose } = \lambda f: \tau_2 \rightarrow \tau_3. \lambda g: \tau_1 \rightarrow \tau_2. \lambda x: \tau_1. f(g(x)) \text{ in } \cdots
  \]

- Notice we are using repeated λ-abstractions to handle multiple arguments

Evaluation for the λ-calculus

- Values are extended to include λ-abstractions \( \lambda x. \ e \):

  \[
  v ::= \cdots \mid \lambda x. \ e
  \]

  (Note: We elide the type annotations when not needed.)

- and the evaluation rules are extended as follows:

  \[
  e \Downarrow v \quad \text{for } L_{\text{Lam}}
  \]

  \[
  \begin{array}{c}
  \lambda x. e \Downarrow \lambda x. e \\
  e_1 \Downarrow \lambda x. e \\
  e_2 \Downarrow v \quad e_1 e_2 \Downarrow v \quad e_1 [v_2/x] \Downarrow v
  \end{array}
  \]

- Note: Combined with \( \text{let} \), this subsumes named functions! We can just define \( \text{let fun} \) as “syntactic sugar”

  \[
  \text{let fun } f(x: \tau) = e_1 \text{ in } e_2 \iff \text{let } f = \lambda x: \tau. e_1 \text{ in } e_2
  \]

Recursive functions

- However, \( L_{\text{Lam}} \) still cannot express general recursion, e.g. the factorial function:

  \[
  \text{let fun } \text{fact}(n : \text{int}) = \\
  \quad \text{if } n == 0 \text{ then } 1 \text{ else } n \times \text{fact}(n - 1) \text{ in } \cdots
  \]

  is not allowed because \( \text{fact} \) is not in scope inside the function body.

- We can’t write it directly as a λ-expression \( \lambda x: \tau. \ e \) either because we don’t have a “name” for the function we’re trying to define inside \( e \).

  - (Technically, we could get around this problem in an untyped version of the lambda calculus...)
Named recursive functions

- In many languages, named function definitions are recursive by default. (C, Python, Java, Haskell, Scala)
- Others explicitly distinguish between nonrecursive and recursive (named) function definitions. (Scheme, OCaml, F#)

```latex
\text{let } \textit{f}(x) = e \quad \text{// nonrecursive:}
\quad \text{// only } x \text{ in scope in } e
\text{let rec } \textit{f}(x) = e \quad \text{// recursive:}
\quad \text{// both } f \text{ and } x \text{ in scope in } e
```

Note: In the untyped $\lambda$-calculus, \textbf{let rec} is definable using a special $\lambda$-term called the $Y$ combinator.

Anonymous recursive functions

- Inspired by $L_{Lam}$, we introduce a notation for anonymous recursive functions:

```latex
e ::= \cdots | \text{rec } \textit{f}(x : \tau_1) : \tau_2. \; e
```

- Idea: $f$ is a local name for the function being defined, and is in scope in $e$, along with the argument $x$.
- We define $L_{Rec}$ to be $L_{Lam}$ extended with \textbf{rec}.
- We can then define \textbf{let rec} as syntactic sugar:

```latex
\text{let rec } f(x) = e_1 \; \text{in} \; e_2
\iff
\text{let } f = \text{rec } f(x : \tau_1) : \tau_2. \; e_1 \; \text{in} \; e_2
```

Note: The outer $f$ is in scope in $e_2$, while the inner one is in scope in $e_1$. The two $f$ bindings are unrelated.

Anonymous recursive functions: typing

- The types of $L_{Rec}$ are the same. We just add one rule:

```latex
\Gamma \vdash e : \tau \text{ for } L_{Rec}
\frac{\Gamma, f : \tau_1 \rightarrow \tau_2, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \text{rec } f(x : \tau_1) : \tau_2. \; e : \tau_1 \rightarrow \tau_2}
```

This says: to typecheck a recursive function,

- bind $f$ to the type $\tau_1 \rightarrow \tau_2$ (so that we can call it as a function in $e$),
- bind $x$ to the type $\tau_1$ (so that we can use it as an argument in $e$),
- typecheck $e$.

Since we use the same function type, the existing function application rule is unchanged.

Anonymous recursive functions: semantics

- Like a $\lambda$-term, a recursive function is a value:

```latex
v ::= \cdots | \text{rec } f(x). \; e
```

- We can evaluate recursive functions as follows:

```latex
\text{e} \Downarrow v \text{ for } L_{Rec}
\frac{\text{rec } f(x). \; e \Downarrow \text{rec } f(x). \; e}{\text{rec } f(x). \; e_1 \Downarrow \text{rec } f(x). \; e_2 \Downarrow v}
\frac{e_1 \Downarrow v_1, e_2 \Downarrow v_2}{e_1 \; e_2 \Downarrow v[\text{rec } f(x). \; e/f, v_2/x] \Downarrow v}
\frac{e_1 \Downarrow v_1}{e_1 \Downarrow v}
```

To apply a recursive function, we substitute the argument for $x$ and the whole \text{rec} expression for $f$. 

Examples

- We can now write, typecheck and run `fact` (you will implement an evaluator for L_{Rec} in Assignment 2 that can do this).
- In fact, L_{Rec} is *Turing-complete* (though it is still so limited that it is not very useful as a general-purpose language).
- (*Turing complete* means: able to simulate any *Turing machine*, that is, any computable function / any other programming language. ITCS covers Turing completeness and computability in depth.)

Mutual recursion

- What if we want to define mutually recursive functions?
- A simple example:
  ```scala
def even(n: Int) = if n == 0 then true else odd(n-1)
def odd(n: Int) = if n == 0 then false else even(n-1)
```

Perhaps surprisingly, we can’t easily do this!
- One solution: generalize `let rec`:
  ```scala
  let rec f1(x1:τ1) : τ1' = e1 and ... and fn(xn:τn) : τn' = en
  in e
  ```
  where f1,..., fn are all in scope in bodies e1,..., en.
- This gets messy fast; we’ll revisit this issue later.

Summary

- Today we have covered:
  - Named functions
  - Static vs. dynamic scope
  - Anonymous functions
  - Recursive functions
- along with our first “composite” type, the function type \( τ_1 \rightarrow τ_2 \).
- Next time
  - Data structures: Pairs (combination) and variants (choice)