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:

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

or (assuming inequality tests) absolute value:

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

$$ \Gamma, x: \tau_1 \vdash e_1: \tau_2 \quad \Gamma, f: \tau_1 \rightarrow \tau_2 \vdash e_2: \tau $$

$$ \Gamma \vdash \text{let fun } f(x: \tau_1) = e_1 \text{ in } e_2: \tau $$

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 $\delta$.

$$ \delta[f \mapsto \langle x \Rightarrow e_1 \rangle], e_2 \Downarrow v $$

$$ \delta, \text{let fun } f(x: \tau) = e_1 \text{ in } e_2 \Downarrow v $$

When we encounter an application, look up the definition and evaluate the body with the argument value substituted for the argument:

$$ \delta, e_0 \Downarrow v \quad \delta(f) = \langle x \Rightarrow e \rangle \quad \delta, e[v_0/x] \Downarrow v $$

$$ \delta, f(e_0) \Downarrow v $$

For convenience, we just use a single environment $\Gamma$ for both variables and function names.

Typechecking of $\mathit{abs}(-42)$

$$ \Gamma(x) = \mathbb{int} $$

$$ \Gamma \vdash x: \mathbb{int} \quad \Gamma \vdash 0: \mathbb{int} $$

$$ \Gamma \vdash x < 0: \mathbb{bool} \quad \Gamma \vdash -x: \mathbb{int} \quad \Gamma \vdash x: \mathbb{int} $$

$$ \Gamma \vdash \text{if } x < 0 \text{ then } -x \text{ else } x: \mathbb{int} $$

$$ \Gamma \vdash e_{\mathit{abs}}: \mathbb{int} $$

$$ \mathit{abs} : \mathbb{int} \rightarrow \mathbb{int} \vdash \mathit{abs}(-42): \mathbb{int} $$

$$ \Gamma \vdash \text{let fun } \mathit{abs}(x: \mathbb{int}) = e_{\mathit{abs}} \text{ in } \mathit{abs}(-42): \mathbb{int} $$

where $\Gamma = x: \mathbb{int}$.

Evaluation of $\mathit{abs}(-42)$

$$ \delta, -42 < 0 \Downarrow \text{true} \quad \delta, -(42) \Downarrow 42 $$

$$ \delta, \text{if } -42 < 0 \text{ then } -(42) \text{ else } -42 \Downarrow 42 $$

$$ \delta, -42 \Downarrow 42 \quad \delta(\mathit{abs}) = \langle x \Rightarrow e_{\mathit{abs}} \rangle $$

$$ \delta, e_{\mathit{abs}}[-42/x] \Downarrow 42 $$

$$ \delta, \mathit{abs}(-42) \Downarrow 42 $$

$$ \text{let fun } \mathit{abs}(x: \mathbb{int}) = e_{\mathit{abs}} \text{ in } \mathit{abs}(-42) \Downarrow 42 $$

where $e_{\mathit{abs}} = \text{if } x < 0 \text{ then } -x \text{ else } x$ and

$$ \delta = [\mathit{abs} \mapsto \langle x \Rightarrow e_{\mathit{abs}} \rangle] $$
**Static vs. dynamic scope**

- What if we do this?
  ```
  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:
  ```
  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:
  ```
  \( \lambda x : \tau. \ e \)
  ```

  Here, `\` (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**

- Consider the following language:
  ```
  e ::= x | e_1 e_2 | \lambda x. \ e
  ```

  i.e. we just have variables, function applications, and lambda-abstractions.

  Application `e_1 e_2` applies a function term to an argument

  This is called the (untyped) \( \lambda \)-calculus

  It can serve as an expressive programming language / computational model on its own.

  (The course “Introduction to Theoretical Computer Science” explores its use as a foundation for computation.)

  We will focus on the typed version.
We define $L_{\text{Lam}}$ to be $L_{\text{Let}}$ extended with typed $\lambda$-abstraction and application as follows:

$e ::= \cdots \mid e_1 e_2 \mid \lambda x:\tau.\ e$

$\tau ::= \cdots \mid \tau_1 \rightarrow \tau_2$

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

We can extend the typing rules as follows:

$\Gamma \vdash e : \tau$ for $L_{\text{Lam}}$

$\Gamma, x:\tau_1 \vdash e : \tau_2$

$\Gamma \vdash \lambda x:\tau_1.\ e : \tau_1 \rightarrow \tau_2$

$\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1$

$\Gamma \vdash e_1 e_2 : \tau_2$

Values are extended to include $\lambda$-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:

$e \Downarrow v$ for $L_{\text{Lam}}$

$\lambda x.\ e \Downarrow \lambda x.\ e$

$e_1 \Downarrow \lambda x.\ e \quad e_2 \Downarrow v_2 \quad e[v_2/x] \Downarrow v$

$\Gamma \vdash e_1 \Downarrow v \quad \Gamma \vdash e_2 \Downarrow v$

$\Gamma \vdash e_1 e_2 \Downarrow v$

Note: Combined with let, this subsumes named functions! We can just define let fun as “syntactic sugar”

$\text{let fun } f(\cdot,\cdot) = e_1 \text{ in } e_2 \iff \text{let } f = \lambda x:\tau.\ e_1 \text{ in } e_2$

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 $\lambda$-abstractions to handle multiple arguments (compare with lab exercise)

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

$\text{let fun } \text{fact}(\cdot:\text{int}) =$

$\quad \text{if } n == 0 \text{ then } 1 \text{ else } n \times \text{fact}(n - 1) \text{ in } \cdots$

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

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

Examples

Recursive functions
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#)

```
let f(x) = e // nonrecursive:  
  // only x is in scope in e
let rec f(x) = e // recursive: 
  // both f and x in scope in e
```

Note: In the untyped \( \lambda \)-calculus, `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:

```
e ::= \cdots | \text{rec } f(x) : \tau_1 \rightarrow \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 \( \text{L\_Rec} \) to be \( \text{L\_Lam} \) extended with `rec`.

We can then define `let rec` as syntactic sugar:

```
let rec f(x : \tau_1) : \tau_2 = e_1 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 \( \text{L\_Rec} \) are the same. We just add one rule:

\[
\Gamma \vdash e : \tau \quad \text{for } \text{L\_Rec}
\]

\[
\frac{\Gamma, f : \tau_1 \rightarrow \tau_2, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \text{rec } f(x) : \tau_2, e : \tau_1 \rightarrow \tau_2}
\]

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:

\[
v ::= \cdots | \text{rec } f(x) . e
\]

- We can evaluate recursive functions as follows:

\[
\text{rec } f(x) . e \Downarrow \text{rec } f(x) . e
\]

\[
e_1 \Downarrow \text{rec } f(x) . e \quad e_2 \Downarrow v_2 \quad e[\text{rec } f(x) . e/f, v_2/x] \Downarrow v
\]

\[
e_1, e_2 \Downarrow v
\]

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\textsubscript{Rec} in CW1, and write other recursive functions)
- In fact, L\textsubscript{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:
  ```
  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 \texttt{let rec}:
  ```
  let rec f\textsubscript{1}(x\textsubscript{1}:\tau\textsubscript{1}) : \tau\textsubscript{1} = e\textsubscript{1} \quad \text{and} \quad f\textsubscript{n}(x\textsubscript{n}:\tau\textsubscript{n}) : \tau\textsubscript{n} = e\textsubscript{n}
  ```
  where \( f\textsubscript{1}, \ldots, f\textsubscript{n} \) are all in scope in bodies \( e\textsubscript{1}, \ldots, e\textsubscript{n} \).
- This gets messy fast; we’ll revisit this issue later.

Big-step vs. small-step

- Recursion highlights some limitations of big-step semantics
- Specifically, it cannot easily distinguish between nontermination
  ```
  let rec f(x) = f(x + 1) in f(0)
  ```
  and failure:
  ```
  1 + true
  ```
  - (Nor is it helpful for computations that are intended to run forever, perhaps performing side-effects along the way.)
  - We will explore an alternative, small-step semantics in future lectures

Summary

- Today we have covered:
  - Named functions
  - Static vs. dynamic scope
  - Anonymous functions
  - Recursive functions
  - along with our first “composite” type, the function type \( \tau\textsubscript{1} \rightarrow \tau\textsubscript{2} \).
- Next time
  - Data structures: Pairs (combination) and variants (choice)