Elements of Programming Languages

Lecture 5: Functions and recursion

James Cheney

University of Edinburgh

October 3, 2024

Overview

- So far, we've covered
  - arithmetic
  - booleans, conditionals (`if then else`)
  - variables and simple binding (`let`)
- L_{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:

$$e ::= \cdots \mid f(e) \mid \texttt{let fun } f(x:\tau) = e_1 \texttt{ in } e_2$$

- 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 only call f, can't pass a function as an argument to another)

Examples

- We can define a squaring function:

$$\texttt{let fun } \mathit{square}(x : \texttt{int}) = x \times x \texttt{ in } \cdots$$

- or (assuming inequality tests) absolute value:

$$\texttt{let fun } \mathit{abs}(x : \texttt{int}) = \texttt{if } x < 0 \texttt{ then } -x \texttt{ else } x \texttt{ in } \cdots$$

Types for named functions

- We introduce a *type constructor* $\tau_1 \to \tau_2$, meaning "the type of functions taking arguments in τ_1 and returning τ_2"
- We can typecheck named functions as follows:

$$\frac{\Gamma, x{:}\tau_1 \vdash e_1 : \tau_2 \quad \Gamma, f{:}\tau_1 \to \tau_2 \vdash e_2 : \tau}{\Gamma \vdash \texttt{let fun } f(x : \tau_1) = e_1 \texttt{ in } e_2 : \tau}$$

$$\frac{\Gamma(f) = \tau_1 \to \tau_2 \quad \Gamma \vdash e : \tau_1}{\Gamma \vdash f(e) : \tau_2}$$

- For convenience, we just use a single environment Γ for both variables and function names.

Example

Typechecking of $abs(-42)$

$$\dfrac{\dfrac{\dfrac{\Gamma(x) = \mathtt{int}}{\Gamma \vdash x : \mathtt{int}} \quad \dfrac{}{\Gamma \vdash 0 : \mathtt{int}}}{\Gamma \vdash x < 0 : \mathtt{bool}} \quad \dfrac{\dfrac{\Gamma(x) = \mathtt{int}}{\Gamma \vdash x : \mathtt{int}}}{\Gamma \vdash -x : \mathtt{int}} \quad \dfrac{\Gamma(x) = \mathtt{int}}{\Gamma \vdash x : \mathtt{int}}}{\Gamma \vdash \mathtt{if}\ x < 0\ \mathtt{then}\ -x\ \mathtt{else}\ x : \mathtt{int}}$$

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash e_{abs} : \mathtt{int}} \quad \dfrac{\dfrac{}{abs{:}\mathtt{int} \to \mathtt{int} \vdash -42 : \mathtt{int}}}{abs{:}\mathtt{int} \to \mathtt{int} \vdash abs(-42) : \mathtt{int}}}{\vdash \mathtt{let\ fun}\ abs(x : \mathtt{int}) = e_{abs}\ \mathtt{in}\ abs(-42) : \mathtt{int}}$$

where $e_{abs} = \mathtt{if}\ x < 0\ \mathtt{then}\ -x\ \mathtt{else}\ x$ and $\Gamma = x{:}\mathtt{int}$.

Semantics of named functions

- We can define rules for evaluating named functions as follows.
- First, let δ 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 δ.

$$\frac{\delta[f \mapsto \langle x \Rightarrow e_1 \rangle], e_2 \Downarrow v}{\delta, \texttt{let fun } f(x : \tau) = e_1 \texttt{ in } e_2 \Downarrow v}$$

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

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

Examples

Evaluation of $abs(-42)$

$$\frac{\delta, -42 < 0 \Downarrow \texttt{true} \quad \delta, -(-42) \Downarrow 42}{\delta, \texttt{if } -42 < 0 \texttt{ then } -(-42) \texttt{ else } -42 \Downarrow 42}$$

$$\dfrac{\dfrac{\delta, -42 \Downarrow -42 \quad \delta(abs) = \langle x \Rightarrow e_{abs} \rangle \quad \dfrac{\vdots}{\delta, e_{abs}[-42/x] \Downarrow 42}}{\delta, abs(-42) \Downarrow 42}}{\texttt{let fun } abs(x : \texttt{int}) = e_{abs} \texttt{ in } abs(-42) \Downarrow 42}$$

where $e_{abs} = \texttt{if } x < 0 \texttt{ then } -x \texttt{ else } x$ and
$\delta = [abs \mapsto \langle x \Rightarrow e_{abs} \rangle]$

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**.

Static vs. dynamic scope

- Function bodies can contain free variables. Consider:

$$\texttt{let } x = 1 \texttt{ in}$$
$$\texttt{let fun } f(y : \texttt{int}) = x + y \texttt{ in}$$
$$\texttt{let } x = 10 \texttt{ in } f(3)$$

- 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:

$$\begin{array}{l} \texttt{let}\ x = 1\ \texttt{in} \\ \texttt{let fun}\ f(y : \texttt{int}) = x + y\ \texttt{in} \\ \texttt{let}\ x = \texttt{true}\ \texttt{in}\ f(3) \end{array}$$

- 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, though present in e.g. JavaScript, Python

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, λ (Greek letter lambda) introduces an anonymous function expression in which x is bound in e.
  - (The λ-notation dates to Church's higher-order logic (1940); there are several competing stories about why λ is used.)
- 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 $\mathsf{L}_{\mathsf{Lam}}$ to be $\mathsf{L}_{\mathsf{Let}}$ extended with typed λ-abstraction and application as follows:

$$\begin{array}{rcl} e & ::= & \cdots \mid e_1\ e_2 \mid \lambda x{:}\tau.\ e \\ \tau & ::= & \cdots \mid \tau_1 \to \tau_2 \end{array}$$

- $\tau_1 \to \tau_2$ is (again) the type of *functions from* τ_1 *to* τ_2.
- We can extend the typing rules as follows:

$\boxed{\Gamma \vdash e : \tau}$ for $\mathsf{L}_{\mathsf{Lam}}$

$$\frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.\ e : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

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:

$\boxed{e \Downarrow v}$ for $\mathrm{L}_{\mathrm{Lam}}$

$$\frac{}{\lambda x.\ e \Downarrow \lambda x.\ e} \qquad \frac{e_1 \Downarrow \lambda x.e \quad e_2 \Downarrow v_2 \quad e[v_2/x] \Downarrow v}{e_1\ e_2 \Downarrow v}$$

- Note: Combined with `let`, this subsumes named functions! We can just define `let fun` as "syntactic sugar"

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

Examples

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

$$\texttt{let fun } \mathit{twice}(f : \tau \to \tau) = \lambda x{:}\tau.\ f(f(x))\ \texttt{in}\ \cdots$$

- and we can define the composition of two functions:

$$\texttt{let } \mathit{compose} = \lambda f{:}\tau_2 \to \tau_3.\ \lambda g{:}\tau_1 \to \tau_2.\ \lambda x{:}\tau_1.\ f(g(x))\ \texttt{in}\ \cdots$$

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

Recursive functions

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

  $$\begin{array}{l}\texttt{let fun } \mathit{fact}(n{:}\texttt{int}) = \\ \qquad \texttt{if } n == 0 \texttt{ then } 1 \texttt{ else } n \times \mathit{fact}(n-1) \texttt{ in } \cdots\end{array}$$

  is not allowed because *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#)

```
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* λ-calculus, `let rec` is *definable* using a special λ-term called the *Y combinator*

Anonymous recursive functions

- Inspired by $\mathsf{L}_{\mathsf{Lam}}$, we introduce a notation for anonymous *recursive* functions:

$$e ::= \cdots \mid \texttt{rec}\ f(x : \tau_1) : \tau_2.\ e$$

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

$$\begin{aligned} &\texttt{let rec}\ f(x{:}\tau_1) : \tau_2 = e_1\ \texttt{in}\ e_2 \\ &\qquad \iff \texttt{let}\ f = \texttt{rec}\ f(x{:}\tau_1) : \tau_2.\ e_1\ \texttt{in}\ e_2 \end{aligned}$$

- 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:

$\boxed{\Gamma \vdash e : \tau}$ for L_{Rec}

$$\frac{\Gamma, f : \tau_1 \to \tau_2, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{rec } f(x{:}\tau_1) : \tau_2.\ e : \tau_1 \to \tau_2}$$

- This says: to typecheck a recursive function,
  - bind f to the type $\tau_1 \to \tau_2$ (so that we can call it as a function in e),
  - bind x to the type τ_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 λ-term, a recursive function is a value:

$$v ::= \cdots \mid \texttt{rec}\ f(x).\ e$$

- We can evaluate recursive functions as follows:

$\boxed{e \Downarrow v}$ for L_{Rec}

$$\frac{}{\texttt{rec}\ f(x).\ e \Downarrow \texttt{rec}\ f(x).\ e}$$

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

- To apply a recursive function, we substitute the argument for x and the whole `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:

```
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`:

  $$\texttt{let rec } f_1(x_1{:}\tau_1) : \tau_1' = e_1 \texttt{ and } \cdots \texttt{ and } f_n(x_n{:}\tau_n) : \tau_n' = e_n \texttt{ in } e$$

  where $f_1, \ldots, f_n$ are all in scope in bodies $e_1, \ldots, e_n$.
- 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 $\tau_1 \rightarrow \tau_2$.
- Next time
    - Data structures: Pairs (combination) and variants (choice)