COMP 321 Lecture 15: Function Types

Dan Licata

Fall 2026

1 First-order functions

In IMP, functions did not return, so the body of a function declaration was a statement. Today, we will consider functions that do return a value. Starting from a basic functional language with variables, we allow function declarations in the syntax of programs, and function application in the syntax of expressions:

$$p ::= \ldots \mid (f(x : \tau_1) : \tau_2 = e)$$
$$e ::= \ldots \mid x \mid fe$$

A function declaration has a function name f, a variable for the argument, a type τ_1 of the argument, and a type τ_2 of the return value. In C-like concrete syntax, we'd write `tau2 f(tau1 x) { e }` and in ML-like syntax we'd write `fun f(x : tau1) : tau2 = e`.

Because we can have product types in the language, it suffices to consider functions with a single input and a single output:

- A function with more than one argument can take a product type $\tau_1 \times \tau_2$ as input.
- A function zero inputs can take the unit type as input.
- A function with multiple outputs can return a product type.
- A function that would "return void" in C can return the unit type as output — having no return value conveys the same information as returning the unique value of unit type.

Here are the typing rules:

$$\frac{x : \tau_1 \vdash e : \tau_2}{(f(x : \tau_1) : \tau_2 = e)\ \mathsf{ok}} \qquad \frac{(f(x : \tau_1) : \tau_2 = \ldots) \in p \quad \Gamma \vdash e : \tau_1}{\Gamma \vdash f(e) : \tau_2}$$

We say that a program p is ok if each declaration in it is ok. A function declaration is ok when the body of the function has the indicated type with the variable in the context.

$$\frac{e \mapsto e'}{fe \mapsto fe'} \qquad \frac{(f(x : \tau_1) : \tau_2 = e_2) \in p \quad e_1\ \mathsf{done}}{fe_1 \mapsto e_2[e_1/x]}$$

It's implicit that we always extend the definition of substitution to each new form of expression by substituting into the subtrees, so

$$f(e)[e_0/x_0] = f(e[e_0/x_0])$$

Using substitution, the operational semantics for function applications says to substitute the actual argument for the variable.

The notion of function that we have considered so far is called *first-order* and *top-level*:

- Functions can only be declared at the top level of the program, so the body of a function has no variables in scope besides the argument (unless we add top-level variable declarations).
- Functions are treated differently from other types — they cannot be input to a function, or output from a function, or stored in a data structure.

2 Higher-order functions

We now lift these restrictions.

We consider the function type $\tau_1 \to \tau_2$ to have the same status as the existing types of the language like int and $\tau_1 \times \tau_2$ and $\tau_1 + \tau_2$:

$$\tau ::= \mathsf{int} \mid \tau_1 \to \tau_2 \mid \tau_1 + \tau_2 \mid \tau_1 \times \tau_2$$

The expressions include function application as well as function values:

$$e ::= \ldots \mid x \mid \mathsf{Lam}(\tau_1, x.e) \mid \mathsf{App}(e_1, e_2)$$

In concrete syntax, we write $\lambda x : \tau_1.e$ or `lam x:T.e` for a function value, and $e_1(e_2)$ for function application. The Greek letter lambda is tradition coming from the origins of higher-order functions in Alonzo Church's λ-calculus.

The typing rules are as follows:

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \mathsf{Lam}(\tau_1, x.e) : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_2 \to \tau \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{App}(e_1, e_2) : \tau}$$

The typing rule for function values is a lot like the rule for function declarations above, except for three things:

- $\lambda x : \tau.e$ is an *anonymous function* like `fn x => e` in SML, so there is no name for the function itself. This can be added handled orthogonally using let:

  $$\mathsf{let}\ f\ = \lambda x : \tau_1.e\ \mathsf{in}\ \ldots$$

  is like the above function declaration.
- We don't explicitly write the return type τ_2, because this can be inferred from e's type. On paper, we will often omit the argument type τ_1, but formally it should be there.
- In the function value, all of the other variables in Γ are still in scope in the body of a function. This is sometimes called a *closure*.

The operational semantics says to run the function position until it is a function value, run the argument until it is done, and then substitute the argument's value for the variable in the function value:

$$\frac{}{\mathsf{Lam}(\tau, x.e)\ \mathsf{done}}$$

$$\frac{e_1 \mapsto e_1'}{\mathsf{App}(e_1, e_2) \mapsto \mathsf{App}(e_1', e_2)}\ \texttt{app-search-1} \qquad \frac{e_2 \mapsto e_2' \quad e_1\ \mathsf{done}}{\mathsf{App}(e_1, e_2) \mapsto \mathsf{App}(e_1, e_2')}\ \texttt{app-search-2}$$

$$\frac{e_2\ \mathsf{done}}{\mathsf{App}(\mathsf{Lam}(\tau, x.e), e_2) \mapsto e[e_2/x]}\ \texttt{app-instr}$$

Our substitution-based operational semantics naturally handles closures without any special treatment. For example, consider

```
(let x = 5 in
  lam y : int. x + y
end) 6
```

By the operational semantics for `let` this steps to

```
(lam y:int.5 + y) 6
```

which then by `app-instr` steps to `5 + 6`.

3 Type Safety

The proof of type safety does not really need any new ideas.

Theorem 1. *Progress If* $\cdot \vdash e : \tau$ *then either* $e\ \mathsf{done}$ *or there exists an* e' *such that* $e \mapsto e'$.

New cases:

- For
$$\frac{x : \tau_1 \vdash e : \tau_2}{\cdot \vdash \mathsf{Lam}(\tau_1, x.e) : \tau_1 \to \tau_2}$$
$\mathsf{Lam}(\tau_1, x.e)$ is done.

- For
$$\frac{\cdot \vdash e_1 : \tau_2 \to \tau \quad \cdot \vdash e_2 : \tau_2}{\cdot \vdash \mathsf{App}(e_1, e_2) : \tau}$$
By the inductive hypothesis, either e_1 is done or it takes a step. If it takes a step, then so does $\mathsf{App}(e_1, e_2)$ by `app-search-1`. If it is done, then by the inductive hypothesis, either e_2 is done or it takes a step. If it takes a step, use `app-search-2`. This is just our usual proof for two-input operations. If both e_1 and e_2 are done, then we have a canonical forms principle that any done expression of function type is a $\mathsf{Lam}(\tau_2, x.e)$. So by `app-instr`,
$$\mathsf{App}(e_1, e_2) = \mathsf{App}(\mathsf{Lam}(\tau_2, x.e), e_2) \mapsto e[e_2/x]$$

Theorem 2. *Preservation If* $e \mapsto e'$ *and* $\cdot \vdash e : \tau$ *then either* $\cdot \vdash e' : \tau$.

New cases:

- $$\frac{e_1 \mapsto e_1'}{\mathsf{App}(e_1, e_2) \mapsto \mathsf{App}(e_1', e_2)}\ \texttt{app-search-1}$$

  Assume $\cdot \vdash \mathsf{App}(e_1, e_2) : \tau$. By inversion $\cdot \vdash e_1 : \tau_2 \to \tau$ and $\cdot \vdash e_2 : \tau_2$ for some type τ_2. So by the IH, $\cdot \vdash e_1' : \tau_2 \to \tau$, and we can reapply the rule to get $\cdot \vdash \mathsf{App}(e_1', e_2) : \tau$.

- $$\frac{e_2 \mapsto e_2' \quad e_1\ \mathsf{done}}{\mathsf{App}(e_1, e_2) \mapsto \mathsf{App}(e_1, e_2')}\ \texttt{app-search-2}$$

  Assume $\cdot \vdash \mathsf{App}(e_1, e_2) : \tau$. By inversion $\cdot \vdash e_1 : \tau_2 \to \tau$ and $\cdot \vdash e_2 : \tau_2$ for some type τ_2. So by the IH, $\cdot \vdash e_2' : \tau_2$, and we can reapply the rule to get $\cdot \vdash \mathsf{App}(e_1, e_2') : \tau$.

- $$\frac{e_2\ \mathsf{done}}{\mathsf{App}(\mathsf{Lam}(\tau, x.e), e_2) \mapsto e[e_2/x]}\ \texttt{app-instr}$$

  Assume $\cdot \vdash \mathsf{App}(\mathsf{Lam}(\tau_2, x.e), e_2) : \tau$. By inversion $\cdot \vdash \mathsf{Lam}(\tau_2, x.e) : \tau_2 \to \tau$ and $\cdot \vdash e_2 : \tau_2$. By inversion again, $x : \tau_2 \vdash e : \tau$. Therefore, by the substitution lemma (see Lecture 12), $\cdot \vdash e[e_2/x] : \tau$.