Functional basics

At this point, we've now laid the foundations for exploring individual programming language features in greater detail. Specifically, building on the base of the simply-typed lambda calculus from last lecture, we can create all the features of a modern Church language by defining its syntax, static semantics, and dynamic semantics. Today, we'll start by defining features that look similar to the kinds you'd find in a standard Turing language like Java or Python, but this time explore their formal representation.

Let binding

In Turing languages, you're used to variable assignment as a *statement*, distinct from expressions. Assigning to a variable implicitly updates some scope-local state that contains a mutable slot for the variable's value. (When you say it that way, "standard" variables seem kind of complicated!) In an expression-oriented language, variable assignment is just another expression. For example:

$$\mathsf{let}\ x : \mathsf{num} = 1\ \mathsf{in}\ x + 1 \overset{*}{\mapsto} 2$$

This expression replaces each instance of x in the "let" body $x + 1$ with the value 1, so the entire expression evalutes to 2. We can stack "let" bindings to produce what looks like a standard straight line program:

$$\begin{array}{l}\mathsf{let}\ x : \mathsf{num} = 1\ \mathsf{in} \\ \mathsf{let}\ y : \mathsf{num} = x + 1\ \mathsf{in} \\ y * 2\end{array}$$

If you replaced the "in" keyword with a semicolon, the syntax starts to look suspiciously familiar... but it's still all one expression!

If we wanted to, the syntax, static semantics, and dynamic semantics of "let" could be defined through a grammar, type judgments, and operational semantics like before. However, our language is now sufficiently complex that we can start defining new features in terms of existing features. This is the idea of *syntactic sugar*, or functionally equivalent notation. For example, `x += 1` is sugar for `x = x + 1` in most languages. Here, we can observe that "let" bindings are actually just sugar for function application!

$$\mathsf{let}\ x : \tau = e_{\mathsf{var}}\ \mathsf{in}\ e_{\mathsf{body}} \triangleq (\lambda\ (x : \tau)\ .\ e_{\mathsf{body}})\ e_{\mathsf{var}}$$

The $\triangleq$ sign means "defined as", which you can informally understand to mean "any usage of the term on the left is equivalent to the term on the right". For "let", we can simulate it by creating a function and immediately calling it with the corresponding value for the variable.

Recursion

In a Turing language, recursion is inseparable from functions. All functions have names, and functions are allowed to use their names to recursively call themselves. However, there's no *a priori* reason why only functions need to be recursive. The essence of recursion boils down to the idea of *self-reference*, that an object has a handle to its self. We capture this idea with a "fix" operator:

$$\begin{array}{rcll}\text{Expression } e ::= & \ldots & & \\ & \mid & \mathsf{fix}\ (x : \tau)\ .\ e & \text{Fixpoint}\end{array}$$

$$\frac{}{\mathsf{fix}\ (x : \tau)\ .\ e \mapsto [x \to \mathsf{fix}\ (x : \tau)\ .\ e]\ e}\ (\text{D-Fix}) \qquad \frac{\Gamma, x : \tau \vdash e : \tau}{\Gamma \vdash \mathsf{fix}\ (x : \tau)\ .\ e : \tau}\ (\text{T-Fix})$$

Essentially, the fix operator says: in an expression e, any reference to the variable x will be replaced with e itself. x becomes the self-referential handle for recursion. For example, we could compute an infinite sum:

$$(\mathsf{fix}\ (n : \mathsf{num})\ .\ 1 + n) \mapsto (1 + (\mathsf{fix}\ (n : \mathsf{num})\ .\ 1 + n)) \mapsto (1 + (1 + (\mathsf{fix}\ (n : \mathsf{num})\ .\ 1 + n))) \mapsto \ldots$$

To type-check a fixpoint, our T-Fix rule says: from the fixpoint expression, the body is supposed to be of type τ from the syntax $x : \tau$. If the body actually has that type given $x : \tau$, then the whole expression also has type τ. For example, the following usage is incorrect:

$$\mathsf{fix}\ (x : \mathsf{num})\ .\ x\ 1$$

If x is supposed to be a number, but we use it as a function, that violates our typing rule.

Next, we can combine the fixpoint operator with a function definition to make a recursive function. This is best captured through a new syntactic sugar for letrec:

$$\mathsf{letrec}\ x : \tau = e_{\mathsf{var}}\ \mathsf{in}\ e_{\mathsf{body}} \triangleq \mathsf{let}\ x : \tau = \mathsf{fix}\ (x : \tau)\ .\ e_{\mathsf{var}}\ \mathsf{in}\ e_{\mathsf{body}}$$

For example, we can now write a factorial function using the boolean operators we will define shortly:

$$\mathsf{letrec}\ \mathsf{fact} : \mathsf{num} \to \mathsf{num} = (\lambda\ \ (n : \mathsf{num})\ .\ \mathsf{if}\ n = 0\ \mathsf{then}\ 1\ \mathsf{else}\ n * (\mathsf{fact}\ (n - 1)))\ \mathsf{in}\ \ldots$$

Conditions

Finally, we want to add control flow into our language with if expressions. Again contrasting with prior experience, you're used to "if" being a statement. For example, in Python:

```
x = None
if some_condition():
  x = then_logic()
else:
  x = else_logic()
```

By contrast, in a Church language, if-expressions have values. We would write the above as:

```
let x = if some_condition() then then_logic() else else_logic() in
...
```

The value of the if-expression is the result of evaluating either the left or right sub-expressions. Formally, we add booleans and ifs to our language:

$$
\begin{array}{rcll}
\text{Expression } e ::= & \ldots & & \\
& \mid & \mathsf{true} & \text{True value} \\
& \mid & \mathsf{false} & \text{False value} \\
& \mid & \mathsf{if}\ e_{\mathsf{cond}}\ \mathsf{then}\ e_{\mathsf{then}}\ \mathsf{else}\ e_{\mathsf{else}} & \text{If expression}
\end{array}
$$

$$
\begin{array}{rcll}
\text{Type } \tau ::= & \ldots & & \\
& \mid & \mathsf{bool} & \text{Boolean type}
\end{array}
$$

$$\frac{}{\mathsf{true\ val}}\ \text{(D-True)} \qquad \frac{}{\mathsf{false\ val}}\ \text{(D-False)} \qquad \frac{}{\mathsf{true} : \mathsf{bool}}\ \text{(T-True)} \qquad \frac{}{\mathsf{false} : \mathsf{bool}}\ \text{(T-False)}$$

$$\frac{}{\mathsf{if\ true\ then}\ e_{\mathsf{then}}\ \mathsf{else}\ e_{\mathsf{else}} \mapsto e_{\mathsf{then}}}\ \text{(D-If-True)} \qquad \frac{}{\mathsf{if\ false\ then}\ e_{\mathsf{then}}\ \mathsf{else}\ e_{\mathsf{else}} \mapsto e_{\mathsf{else}}}\ \text{(D-If-False)}$$

$$\frac{e_{\mathsf{cond}} \mapsto e'_{\mathsf{cond}}}{\mathsf{if}\ e_{\mathsf{cond}}\ \mathsf{then}\ e_{\mathsf{then}}\ \mathsf{else}\ e_{\mathsf{else}} \mapsto \mathsf{if}\ e'_{\mathsf{cond}}\ \mathsf{then}\ e_{\mathsf{then}}\ \mathsf{else}\ e_{\mathsf{else}}}\ \text{(D-If-Step)}$$

$$\frac{\Gamma \vdash e_{\mathsf{cond}} : \mathsf{bool} \quad \Gamma \vdash e_{\mathsf{then}} : \tau \quad \Gamma \vdash e_{\mathsf{else}} : \tau}{\Gamma \vdash \mathsf{if}\ e_{\mathsf{cond}}\ \mathsf{then}\ e_{\mathsf{then}}\ \mathsf{else}\ e_{\mathsf{else}} : \tau}\ \text{(T-If)}$$

The dynamic semantics are straightforward. True and false are values, and an if expression returns the "then" if its condition is true, and returns the "else" if the condition is false. Look carefully at the T-If rule for typechecking if-expressions. The condition must evaluate to a boolean. The "then" and "else" branches can evaluate to any type, *but they must evaluate to the same type*. That's implicit by the use of the same symbol τ.

> Note that T-If is *sound*, but *incomplete*. Meaning that from the perspective of type-safety, any expression that satisfies this rule is safe (soundness), but there are expressions that don't satisfy this rule that are still safe (incomplete). For example:
>
> $$\mathsf{let}\ x : \mathsf{num} = (\mathsf{if\ false\ then\ true\ else}\ 0)\ \mathsf{in}\ \ldots$$
>
> This expression will always evaluate to 0, so any downstream logic depending on x will never enter a stuck state. However, special-casing on these kinds of examples is tedious, non-exhaustive, and inconsisent from a language design perspective. So we prefer sound rules that are simple and understandable over too many extra cases.

A boolean language of only true and false isn't interesting, so we will add the standard relational and logical operators on numbers.

$$
\begin{array}{rcl}
\text{RelOp } \bowtie ::= & < \mid > \mid = &
\end{array}
$$

$$
\begin{array}{rcll}
\text{Expression } e ::= & \ldots & & \\
& \mid & e_L \bowtie e_R & \text{Relational operators} \\
& \mid & e_L \wedge e_R & \text{Boolean and} \\
& \mid & e_L \vee e_R & \text{Boolean or}
\end{array}
$$

It's a useful exercise to try and define these rules for yourself. This is essentially an operationalized version of the schema checking you did in Assignment 1.

First, the dynamic semantics:

$$\frac{n' = n_L \bowtie n_R}{n_L \bowtie n_R \mapsto n'} \text{ (D-Relop-Op)}$$

$$\frac{}{\mathsf{true} \wedge \mathsf{true} \mapsto \mathsf{true}} \text{ (D-And-T)} \quad \frac{e \text{ val}}{\mathsf{false} \wedge e \mapsto \mathsf{false}} \text{ (D-And-F1)} \quad \frac{e \text{ val}}{e \wedge \mathsf{false} \mapsto \mathsf{false}} \text{ (D-And-F2)}$$

$$\frac{}{\mathsf{false} \vee \mathsf{false} \mapsto \mathsf{false}} \text{ (D-Or-F)} \quad \frac{e \text{ val}}{\mathsf{true} \vee e \mapsto \mathsf{true}} \text{ (D-Or-T1)} \quad \frac{e \text{ val}}{e \vee \mathsf{true} \mapsto \mathsf{true}} \text{ (D-Or-T2)}$$

$$\frac{e_L \mapsto e'_L}{e_L \bowtie e_R \mapsto e'_L \bowtie e_R} \text{ (D-Relop-L)} \quad \frac{e_L \text{ val} \quad e_R \mapsto e'_R}{e_R \bowtie e'_R \mapsto e_L \bowtie e'_R} \text{ (D-Relop-R)}$$

$$\frac{e_L \mapsto e'_L}{e_L \wedge e_R \mapsto e'_L \wedge e_R} \text{ (D-And-L)} \quad \frac{e_L \text{ val} \quad e_R \mapsto e'_R}{e_R \wedge e'_R \mapsto e_L \wedge e'_R} \text{ (D-And-R)}$$

$$\frac{e_L \mapsto e'_L}{e_L \vee e_R \mapsto e'_L \vee e_R} \text{ (D-Or-L)} \quad \frac{e_L \text{ val} \quad e_R \mapsto e'_R}{e_R \vee e'_R \mapsto e_L \vee e'_R} \text{ (D-Or-R)}$$

> You may start to get exhausted by all the semantic rules, particularly the step rules defining evaluation order on expressions. So did the rest of the programming language community! Most formal language semantics these days are written using evaluation contexts to separate the "interesting" rules (i.e. the core semantics) from the "boring" evaluation order rules. We won't cover evaluation contexts in this course due to time, but just recognize that it doesn't have to be this way!

And then the static semantics:

$$\frac{\Gamma \vdash e_L : \mathsf{num} \quad \Gamma \vdash e_R : \mathsf{num}}{\Gamma \vdash e_L \bowtie e_R : \mathsf{bool}} \text{ (T-Relop)} \quad \frac{\Gamma \vdash e_L : \mathsf{bool} \quad \Gamma \vdash e_R : \mathsf{bool}}{\Gamma \vdash e_L \wedge e_R : \mathsf{bool}} \text{ (T-And)} \quad \frac{\Gamma \vdash e_L : \mathsf{bool} \quad \Gamma \vdash e_R : \mathsf{bool}}{\Gamma \vdash e_L \vee e_R : \mathsf{bool}} \text{ (T-Or)}$$

Lambda calculus ↔ OCaml

As mentioned in last lecture's notes, the typed lambda calculus we've described so far maps almost 1-to-1 onto OCaml. You just have to map these syntax changes:

- $\lambda(x : \mathsf{num}).\, e$ becomes `fun (x : int) -> e`
- letrec becomes `let rec`
- $\wedge$ becomes `&&`
- $\vee$ becomes `||`

That's it! For example, we can take our factorial example:

$$\mathsf{letrec\ fact} : \mathsf{num} \to \mathsf{num} = (\lambda\ (n : \mathsf{num})\ .\ \mathsf{if}\ n = 0\ \mathsf{then}\ 1\ \mathsf{else}\ n * (\mathsf{fact}\ (n - 1)))\ \mathsf{in}\ \ldots$$

And turn it into OCaml:

```
$ ocaml
# let rec fact : int -> int = fun (n : int) -> if n = 0 then 1 else n * (fact (n -1)) ;;
val fact : int -> int = <fun>
# fact 5 ;;
- : int = 120
```

(The `;;` is used to delineate expressions sent to the interpreter.)

To check your understanding, try implementing the following functions:

- `max : int -> int -> int` : return the maximum of two numbers.
- `linear_solve : float -> float -> float -> float -> float` : given two points x_1, y_1, x_2, y_2, return the slope of the line $f(x) = ax + b$ through the two points.
- `is_prime : int -> bool` : returns true if the number is prime. (You will need to think about defining helper functions!)