Lecture 3.2 - Lambda calculus

We defined in class a simple language for arithmetic with booleans. The grammar is defined here:

$$
\begin{array}{rll}
\mathsf{Type}\ \tau ::= & \texttt{int} & \text{integer} \\
& \texttt{bool} & \text{boolean} \\
\\
\mathsf{Term}\ t ::= & z & \text{zero} \\
& s(t) & \text{successor} \\
& \texttt{true} & \text{constant true} \\
& \texttt{false} & \text{constant false} \\
& \texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 & \text{if expression} \\
& \texttt{iszero}(t) & \text{zero check}
\end{array}
$$

The grammar defines how you can write down terms in the language. For example, the following are syntactically valid terms:

$$
\begin{array}{l}
z \\
s(s(z)) \\
\texttt{if false then}\ z\ \texttt{else}\ s(z) \\
\texttt{iszero}(s(z)) \\
s(\texttt{false})
\end{array}
$$

Note that the last term is *syntactically* valid—it matches the specification given by the grammar, but it is not *semantically* valid—$s(\texttt{false})$ doesn't make sense since we can't take the successor of false.

Now we're able to define things in a language, but we can't do anything with them. The "doing" that we're interested in is called *dynamics*, which defines how we can step-by-step reduce our terms to their simplest possible forms, called *values*. We will formally define what counts as a value as follows:

$$
\frac{}{z\ \mathsf{val}}\ (\text{V-z}) \qquad \frac{t\ \mathsf{val}}{s(t)\ \mathsf{val}}\ (\text{V-s}) \qquad \frac{}{\texttt{false}\ \mathsf{val}}\ (\text{V-}\textsc{false}) \qquad \frac{}{\texttt{true}\ \mathsf{val}}\ (\text{V-}\textsc{true})
$$

These things are logical *rules* that define an implication—if what's above the bar is true, then what's below the bar is true. Here, val is an example of a *judgment*, or an assertion of truth. So $z\ \mathsf{val}$ reads "z is a value", and the V-s rule reads "if t is a value, then $s(t)$ is a value." The strings in parentheses are the names of the rules, useful for referring back to them in proofs.

Now, we know where our terms start (derived from our grammar) and where they end (as values). Next we need to formally define how we reduce terms down to values, which we define using a *small-step semantics*.

$$\frac{t \mapsto t'}{s(t) \mapsto s(t')}\ (\text{D-s}) \qquad \frac{t_1 \mapsto t_1'}{\texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 \mapsto \texttt{if}\ t_1'\ \texttt{then}\ t_2\ \texttt{else}\ t_3}\ (\text{D-}\textsc{if}_1)$$

$$\frac{}{\texttt{if true then}\ t_2\ \texttt{else}\ t_3 \mapsto t_2}\ (\text{D-}\textsc{if}_2) \qquad \frac{}{\texttt{if false then}\ t_2\ \texttt{else}\ t_3 \mapsto t_3}\ (\text{D-}\textsc{if}_3)$$

$$\frac{t \mapsto t'}{\texttt{iszero}(t) \mapsto \texttt{iszero}(t')}\ (\text{D-}\textsc{iszero}_1) \qquad \frac{}{\texttt{iszero}(z) \mapsto \texttt{true}}\ (\text{D-}\textsc{iszero}_2)$$

$$\frac{}{\texttt{iszero}(s(t)) \mapsto \texttt{false}}\ (\text{D-}\textsc{iszero}_3)$$

Here's an example of using those rules to evaluate a term:

$$\begin{aligned} &\texttt{if iszero}(z)\ \texttt{then}\ s(z)\ \texttt{else}\ z \\ \mapsto\ &\texttt{if true then}\ s(z)\ \texttt{else}\ z && (\text{D-iszero}_1) \\ \mapsto\ &s(z) && (\text{D-}\texttt{if}_2) \end{aligned}$$

Lastly, we need a way to eliminate terms that are syntactically valid but semantically invalid, like $s(\texttt{false})$. To do that, we define a type system, also called the *statics* of the language.

$$\frac{}{z : \texttt{int}}\ (\text{T-z}) \qquad \frac{t : \texttt{int}}{s(t) : \texttt{int}}\ (\text{T-s}) \qquad \frac{}{\texttt{true} : \texttt{bool}}\ (\text{T-}\textsc{true}) \qquad \frac{}{\texttt{false} : \texttt{bool}}\ (\text{T-}\textsc{false})$$

$$\frac{t_1 : \texttt{bool} \qquad t_2 : \tau \qquad t_3 : \tau}{\texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 : \tau}\ (\text{T-}\textsc{if}) \qquad \frac{t : \texttt{int}}{\texttt{iszero}(t) : \texttt{bool}}\ (\text{T-}\textsc{iszero})$$

We can use these typing rules to determine whether a term is *well-typed* (i.e. has a type) or not. For example, here's a derivation of the type of the term used in the previous example.

$$\frac{\dfrac{\dfrac{}{z : \texttt{int}}\ (\text{T-z})}{\texttt{iszero}(z) : \texttt{bool}}\ (\text{T-}\textsc{iszero}) \qquad \dfrac{\dfrac{}{z : \texttt{int}}\ (\text{T-z})}{s(z) : \texttt{int}}\ (\text{T-s}) \qquad \dfrac{}{z : \texttt{int}}\ (\text{T-z})}{\texttt{if iszero}(z)\ \texttt{then}\ s(z)\ \texttt{else}\ z : \texttt{int}}\ (\text{T-}\textsc{if})$$

Now we've fully defined our arithmetic language! It has a grammar, static, and dynamics. However, how can we know for sure that we haven't messed up in our definitions? We want to show that our language is *sound*, or that when we have a well typed term t, that we can always evaluate it to a value with no unexpected behavior or hangs. We can formulate this idea in two theorems.

Progress: if $t : \tau$ then either t val or $\exists t'$ such that $t \mapsto t'$.
Preservation: if $t : \tau$ and $t \mapsto t'$ then $t' : \tau$.

To prove these theorems, we will induct over the possible derivations of τ, i.e. if we show that for all ways that you can make a term with a type that the theorem holds, then it holds for any well-typed term.

Proof. First, we will prove progress by induction on the typing rules.

- (T-z): if $z : \texttt{int}$ then either z val or $\exists t'.$ such that $z \mapsto t'$.
  By (V-z), we know z val.
  The case for `true` and `false` follow by the same logic.

- (T-s): if $s(t) : \texttt{int}$ then either $s(t)$ val or $\exists t'.$ such that $s(t) \mapsto t'$.
  By (T-s), we know that $t : \texttt{int}$.
  By the inductive hypothesis (IH), either t val or $\exists t''$ such that $t \mapsto t''$. Two cases:

  1. t val: by (V-s), then $t \ \mathsf{val} \implies s(t) \ \mathsf{val}$.

  2. $\exists t''$ such that $t \mapsto t''$: by (D-s), then $s(t) \mapsto s(t'')$.
     Theorem holds for $t' = s(t'')$.

- (T-`if`): if $\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 : \tau$ then either $\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3$ val or $\exists t'.$ such that $\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \mapsto t'$.
  By (T-`if`), we know that $t_1 : \texttt{bool}$, $t_2 : \tau$, and $t_3 : \tau$.
  By the IH, either t_1 val or $t_1 \mapsto t_1'$. Two cases:

  1. t_1 val: by typing inversion for `bool`, $t_1 \ \mathsf{val} \land t_1 : \texttt{bool} \implies t_1 = \texttt{false} \lor t_1 = \texttt{true}$.
     If $t_1 = \texttt{false}$, then by (D-$\texttt{if}_3$) then $\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \mapsto t_3$.
     If $t_1 = \texttt{true}$, then by (D-$\texttt{if}_2$) then $\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \mapsto t_2$.

  2. $t_1 \mapsto t_1'$: then by (D-$\texttt{if}_1$) then $\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \mapsto \texttt{if } t_1' \texttt{ then } t_2 \texttt{ else } t_3$.

- (T-`iszero`): if $\texttt{iszero}(t) : \texttt{bool}$ then either $\texttt{iszero}(t)$ val or $\exists t'.$ such that $\texttt{iszero}(t) \mapsto t'$.
  By (T–`isz`), we know that $t : \texttt{int}$.
  By the IH, either t val or $\exists t''$ such that $t \mapsto t''$. Two cases:

  1. t val: by typing inversion for `int`, $t \ \mathsf{val} \land t : \texttt{int} \implies t = z \lor t = s(\_)$.
     If $t = z$ then by (D-$\texttt{iszero}_2$) then $\texttt{iszero}(z) \mapsto \texttt{true}$.
     If $t = s(\_)$ then by (D-$\texttt{iszero}_3$) then $\texttt{iszero}(s(\_)) \mapsto \texttt{false}$.

  2. $t \mapsto t''$: by (D-$\texttt{iszero}_1$) then $\texttt{iszero}(t) \mapsto \texttt{iszero}(t'')$.

The theorem holds for every rule, so the theorem is true for all $t : \tau$.

□

Proof. Next, we will prove preservation, again by induction on the typing rules.

- (T-z), (T-`false`), (T-`true`): z, `true`, and `false` are all values, so they cannot step and the theorem is vacuously true.
- (T-`if`): if `if` t_1 `then` t_2 `else` $t_3 : \tau$ and `if` t_1 `then` t_2 `else` $t_3 \mapsto t'$ then $t' : \tau$.
  By (T-`if`), $t_1 :$ `bool`, $t_2 : \tau$ and $t_3 : \tau$.
  There are three cases where `if` t_1 `then` t_2 `else` t_3 can step:
  1. (D-$\texttt{if}_1$): if $t_1 \mapsto t_1'$, then by the IH, $t_1' :$ `bool`.
     By (T-`if`), then `if` t_1' `then` t_2 `else` $t_3 : \tau$.
  2. (D-$\texttt{if}_2$): if $t_1 =$ `true`, then `if` t_1 `then` t_2 `else` $t_3 \mapsto t_2$.
     We know $t_2 : \tau$, so the theorem holds.
     The equivalent logic holds for (D-$\texttt{if}_3$).
- (T-`iszero`): if $\texttt{iszero}(t) :$ `bool` and $\texttt{iszero}(t) \mapsto t'$ then $t' :$ `bool`.
  By (T-`iszero`), $t :$ `int`.
  There are three cases where $\texttt{iszero}(t)$ can step:
  1. (D-$\texttt{iszero}_1$): if $t \mapsto t''$ then by the IH, $t'' :$ `int`.
     By (T-`iszero`), then $\texttt{iszero}(t'') :$ `bool`.
  2. (D-$\texttt{iszero}_2$): if $t = z$, then $\texttt{iszero}(z) \mapsto$ `true`.
     By (T-`true`), `true` : `bool`.
     The equivalent logic holds for (D-$\texttt{iszero}_3$).

The theorem holds for every rule, so the theorem holds for all $t : \tau$. □

Lastly, we discussed the simply typed lambda calculus, or a language of numbers and functions. It has the following grammar:

$$
\begin{array}{lll}
\textsf{Type } \tau ::= & \texttt{int} & \text{integer} \\
 & \tau_1 \to \tau_2 & \text{function} \\
 & & \\
\textsf{Term } t ::= & n & \text{numbers} \\
 & t_1 + t_2 & \text{addition} \\
 & x & \text{variable} \\
 & \lambda\ (x : \tau)\ .\ t' & \text{function definition} \\
 & t_1\ t_2 & \text{function application}
\end{array}
$$

We can write functions in a similar style to the OCaml code we saw on monday. Here are a few terms that are syntactically and semantically valid:

$$
\begin{array}{l}
3 \\
3 + 2 \\
\lambda\ (x : \texttt{int})\ .\ x + 1 \\
(\lambda\ (x : \texttt{int} \to \texttt{int})\ .\ x\ 1)\ (\lambda\ (y : \texttt{int})\ .\ y + 1)
\end{array}
$$

We can define its values:

$$
\frac{}{n\ \textsf{val}}\ \text{(V-n)} \qquad \frac{}{\lambda\ (x : \tau)\ .\ t\ \textsf{val}}\ \text{(V-fn)}
$$

And we can define its dynamics:

$$
\frac{t_1 \mapsto t_1'}{t_1\ t_2 \mapsto t_1'\ t_2}\ \text{(D-app}_1\text{)} \qquad \frac{}{(\lambda\ (x : \tau)\ .\ t_1)\ t_2 \mapsto [x \to t_2]\ t_1}\ \text{(D-app}_2\text{)}
$$

$$
\frac{t_1 \mapsto t_1'}{t_1 + t_2 \mapsto t_1' + t_2}\ \text{(D-add}_1\text{)} \qquad \frac{t_1\ \textsf{val} \qquad t_2 \mapsto t_2'}{t_1 + t_2 \mapsto t_1 + t_2'}\ \text{(D-add}_2\text{)} \qquad \frac{n_3 = n_1 + n_2}{n_1 + n_2 \mapsto n_3}\ \text{(D-add}_3\text{)}
$$

Note that we introduce a new concept here: the *substitution* operator. Now that our language has variables introduced by functions, we need the ability to substitute them with actual values. The notation $[x \to b]\ a$ means "replace all instances of x with b in a. See the assignment 3 handout for a few more examples of substitution.

Lastly, we need to define the statics:

$$
\frac{}{n : \texttt{int}}\ \text{(T-n)} \qquad \frac{\Gamma \vdash t_1 : \texttt{int} \qquad \Gamma \vdash t_2 : \texttt{int}}{\Gamma \vdash t_1 + t_2 : \texttt{int}}\ \text{(T-add)} \qquad \frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda\ (x : \tau_1)\ .\ t : \tau_1 \to \tau_2}\ \text{(T-fn)}
$$

$$
\frac{\Gamma \vdash t_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash (\lambda\ (x : \tau_1)\ .\ t_1)\ t_2 : \tau_2}\ \text{(T-app)} \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ \text{(T-var)}
$$

Again, our statics have some new machinery due to the introduction of variables. Specifically, the problem we need to solve is that when we typecheck the body of a function, e.g. $\lambda\ (x : \texttt{int})\ .\ x + 1$, then we need to know when typechecking $x + 1$ that x is an integer. We codify this idea using a *typing context*, or Γ, that is a mapping from variables to their types. The syntax used here to add to the typing context is the comma, so $\Gamma, x : \tau$ is another way of writing $\Gamma \cup \{x : \tau\}$. For example, here is a typing derivation involving the context:

$$\dfrac{\dfrac{\dfrac{x : \texttt{int} \in \{x : \texttt{int}\}}{\{x : \texttt{int}\} \vdash x : \texttt{int}}\ (\text{T-var}) \qquad \dfrac{}{\{x : \texttt{int}\} \vdash 1 : \texttt{int}}\ (\text{T-n})}{\{x : \texttt{int}\} \vdash x + 1 : \texttt{int}}\ (\text{T-add})}{\emptyset \vdash \lambda\ (x : \texttt{int})\ .\ x + 1 : \texttt{int} \to \texttt{int}}\ (\text{T-fn})$$