Scribe: Stephen Chong, Prakash Linga, Andrew Myers Lecturer: Andrew Myers

## 1 Denotational Semantics for lambda calculus

Before we can give the denotational semantics for Lambda-calculus with types, we need to define a new meaning function $\mathcal{T}$. The function $\mathcal{T}$ takes a type $\tau$, and returns the domain associated with that type. For this type system, we can define $\mathcal{T}$ in a straightforward way:

$$\begin{aligned} \mathcal{T}[\![\mathsf{int}]\!] &= \mathbb{Z} \\ \mathcal{T}[\![\mathsf{bool}]\!] &= \{\mathsf{true}, \mathsf{false}\} \\ \mathcal{T}[\![1]\!] &= \mathbb{U} \\ \mathcal{T}[\![\tau_1 \to \tau_2]\!] &= \mathcal{T}[\![\tau_1]\!] \to \mathcal{T}[\![\tau_2]\!] \end{aligned}$$

Note that $\mathcal{T}$ returns an entire domain corresponding to a type, not just an element of a domain. For the semantics of $\lambda^{\to}$, the domains will not need to have any ordering properties: they are just sets. So we have $\mathcal{T}[\![\cdot]\!] \in \mathit{Type} \to \mathit{Set}$. If a program $e$ can be given type $\tau$, then we expect that the denotation of $e$ is an element of $\mathcal{T}[\![\tau]\!]$. Formally,

$$\vdash e : \tau \Rightarrow \mathcal{C}[\![e]\!]\rho_0 \in \mathcal{T}[\![\tau]\!]$$

It only makes sense to ask what the meaning of an expression $e$ is in a context in which the values of the free variables correspond to the types of free variables against which $e$ was type-checked. Therefore, we need to establish a constraint on what environments $\rho$ we use, based on the typing context $\Gamma$. We say that the environment $\rho$ satisfies a typing context $\Gamma$, written as $\rho \models \Gamma$, if for every variable-type mapping $x \mapsto \tau$ in $\Gamma$, $\rho(x)$ is in the domain corresponding to $\tau$. That is,

$$\rho \models \Gamma \stackrel{def}{\Leftrightarrow} \forall x \in \mathit{dom}(\Gamma)\ .\ \rho(x) \in \mathcal{T}[\![\Gamma(x)]\!]$$

Finally, we need to modify the notation we use for the meaning function $\mathcal{C}$ to account for the presence of types. It only makes sense to take the meaning of well-formed (i.e., well-typed) expressions. A convenient way to ensure this is to change the meaning function to map *typing derivations* to meanings rather than *expressions*. Instead of writing $\mathcal{C}[\![e]\!]\rho$, we will use the notation $\mathcal{C}[\![\Gamma \vdash e : \tau]\!]\rho$, where $e$ has the type $\tau$ in the context $\Gamma$. We will not write the entire typing derivation inside the semantic brackets, but it is implied. Therefore the expression $e$ must be well-formed.

This kind of semantics—where we do not ascribe any meaning to ill-formed terms—is known as a Church-style semantics. Semantics follow typing. The alternative is a Curry-style semantics, where we give a meaning even to programs that are not well-typed. Our operational semantics, for example, talks about how even ill-typed terms evaluate.

We expect that our denotational model satisfies the following soundness condition for all $\rho, \Gamma, e, x, \tau$:

$$\rho \models \Gamma \ \wedge\ \Gamma \vdash e : \tau \quad \Rightarrow \quad \mathcal{C}[\![\Gamma \vdash e : \tau]\!]\rho \in \mathcal{T}[\![\tau]\!]$$

We are now ready to give the long-awaited denotational semantics for typed lambda calculus expressions:

$$\begin{aligned} \mathcal{C}[\![\Gamma \vdash n : \mathsf{int}]\!]\rho &= n \\ \mathcal{C}[\![\Gamma \vdash \mathsf{true} : \mathsf{bool}]\!]\rho &= \mathsf{true} \\ \mathcal{C}[\![\Gamma \vdash \mathsf{unit} : 1]\!]\rho &= \mathsf{unit} \\ \mathcal{C}[\![\Gamma[x \mapsto \tau] \vdash x : \tau]\!]\rho &= \rho(x) \\ \mathcal{C}[\![\Gamma \vdash e_0\ e_1 : \tau']\!]\rho &= (\mathcal{C}[\![\Gamma \vdash e_0 : \tau \to \tau']\!]\rho)\ (\mathcal{C}[\![\Gamma \vdash e_1 : \tau]\!]\rho) \\ \mathcal{C}[\![\Gamma \vdash (\lambda x{:}\tau.\, e) : \tau \to \tau')]\!]\rho &= \lambda v \in \mathcal{T}[\![\tau]\!]\,.\,\mathcal{C}[\![\Gamma[x \mapsto \tau] \vdash e : \tau']\!]\rho[x \mapsto v] \end{aligned}$$

Note that the definition $\mathcal{C}[\![\cdot]\!]$ is well-founded; it is defined by induction on the structure of the typing derivation that is its argument. By construction, we can see that $\mathcal{C}[\![\cdot]\!]$ satisfies the soundness condition given above. The only interesting step in this proof by induction on the typing derivation is the case of a lambda term, which requires that we observe that:

$$\rho \models \Gamma \wedge v \in \mathcal{T}[\![\tau]\!] \Rightarrow \rho[x \mapsto v] \models \Gamma[x \mapsto \tau]$$

Note that $\bot$ does not appear anywhere within this semantics for typed lambda calculus. We didn't need to introduce it because we never took a fixed point. Therefore, we conclude that all typed lambda calculus programs terminate, at least according to this model. Of course, we'll want to see that the operational semantics are adequate with respect to this model to ensure that our evaluation relation can actually find the meanings that this denotational model advertises are there.

Another question we might naturally ask is whether our operational semantics and our denotational semantics agree with one another. For example, we can express the idea that they agree on integers using the following adequacy condition:

$$\vdash e : \mathsf{int} \Rightarrow ((e \longrightarrow^* n) \Leftrightarrow (\mathcal{C}[\![\vdash e : \mathsf{int}]\!]\emptyset = n))$$

This can be proved using the technique of logical relations, which we'll see soon.

## 2 Soundness from the Operational Perspective

We will now look at the soundness of the $\lambda^{\rightarrow}$ typing rules from the operational perspective. What this means is,

$$\textit{Typing rules are sound} \iff \textit{no well-formed program gets stuck.}$$

Note that we use well-formed and well-typed equivalently for this language. To be more precise,

$$\vdash e : \tau \ \wedge \ e \longrightarrow^* e' \quad \Rightarrow \quad e' \in \mathit{Value} \vee \exists e''.e' \longrightarrow e''$$

To show this we will use a 2-step approach. We will show the following two lemmas:

- *Preservation*: As we evaluate a program its type is preserved at each step.

$$\vdash e : \tau \ \wedge \ e \longrightarrow e' \Rightarrow \ \vdash e' : \tau$$

- *Progress*: Every program is either a value or can be stepped into another program (and using *preservation* lemma this will be of the same type!)

$$\vdash e : \tau \Rightarrow e = v \in \mathit{Value} \vee \exists e''.e \longrightarrow e''$$

Note that with these two lemmas soundness easily follows. We use induction on the number of steps taken in $e \longrightarrow^* e'$ to show that $e'$ must have the same type as $e$; the *progress* lemma can then be applied to $e$! We will now set out to prove these lemmas.

### 2.1 Proof of Preservation Lemma

Assuming $e : \tau \wedge e \longrightarrow e'$ we need to show that $\vdash e' : \tau$. We will do this by well-founded induction on typing derivations. (Note that the typing derivations are finite and therefore the relation of subderivation is well-founded.) The property we are trying to show on typing derivations is:

$$P(\vdash e : \tau) \iff \forall e'.e \longrightarrow e' \Rightarrow \vdash e' : \tau$$

Given that we know $e \longrightarrow e'$, there are three possibilities corresponding to the three evaluation rules:

$$\frac{e_0 \longrightarrow e_0'}{e_0 \ e_1 \longrightarrow e_0' e_1} \ (L) \quad \frac{e_1 \longrightarrow e_1'}{v_0 \ e_1 \longrightarrow v_0 e_1'} \ (R) \quad \frac{}{(\lambda x{:}\tau.\, e) \ v \longrightarrow e\{v/x\}} \ (\beta)$$

- Case (L): $e = e_0\ e_1$.

  Because we have a typing derivation for $e_0\ e_1$, we know that there are typing derivations for $e_0$ too. We must have $\vdash e_0 : \tau_1 \to \tau$ and $\vdash e_1 : \tau_1$ for some type $\tau_1$. Because the typing derivation for $e_0$ is a subderivation of the typing derivation, by the induction hypothesis the step $e_0 \longrightarrow e_0'$ also preserves type. So we know $\vdash e_0' : \tau_1 \to \tau$, and we now have both of the premises necessary to construct the proof that $e_0'\ e_1$ has the desired type $\tau$.

- Case (R): $e = v_0\ e_1$

  This case is symmetrical to case (L); in this case $e_1$ is the subexpression making the step.

- Case 3: $e = (\lambda x{:}\tau_1.\, e_0')\ v_1$

  The typing derivation of $\vdash e : \tau$ must look like this:

$$\dfrac{\dfrac{x{:}\tau_1 \vdash e_0' : \tau}{\vdash (\lambda x{:}\tau_1.\, e_0') : \tau_1 \to \tau} \quad \vdash v_1 : \tau_1}{\vdash (\lambda x{:}\tau_1.\, e_0')\ v_1 : \tau}$$

  We know that $e \longrightarrow e_0'\{v_1/x\}$, so we need to show that $e_0\{v_1/x\} : \tau$ using the facts $x{:}\tau_1 \vdash e_0' : \tau$ and $\vdash v_1 : \tau_1$. Our induction hypothesis doesn't help us here; we need to prove this separately. It follows as a special case of a *Substitution Lemma* which captures the type preservation of $\beta$-reduction.

## 2.2 *Substitution* lemma

$$\Gamma, x{:}\tau' \vdash e : \tau \ \wedge\ \vdash v : \tau' \Rightarrow \Gamma \vdash e\{v/x\} : \tau$$

We will prove this by induction on the typing derivation of $e$.

- Case 1: $e = b$. In this case the result trivially holds.

- Case 2: a variable

  - Case 2a: $e = x$. Then $e\{v/x\} = v$ and $\tau' = \tau$ and hence the result holds in this case. We have $\vdash v : \tau$ and need $\Gamma \vdash v : \tau$. This holds because adding new things to the typing context doesn't cause us to lose the ability to produce a typing derivation. This property can be expressed in the following *weakening lemma* whose proof we leave as an exercise:

$$\Gamma \vdash e : \tau \ \wedge\ x \notin \mathrm{dom}(\Gamma) \Rightarrow \Gamma, x{:}\tau' \vdash e : \tau$$

    Then we can get from $\emptyset \vdash v : \tau$ to $\Gamma \vdash v : \tau$ by applying the weakening lemma enough times to add in the finite number of bindings in $\Gamma$.

  - Case 2b: $e = y \neq x$. In this case $e\{v/x\} = e$ and therefore the result trivially holds.

- Case 3: $e = e_0\ e_1$. Using the definition of substitution, $e\{v/x\} = e_0\{v/x\}\ e_1\{v/x\}$.

  Using the derivation of $\Gamma, x{:}\tau' \vdash e : \tau$ we have $\Gamma, x{:}\tau' \vdash e_0 : \tau_1 \to \tau$ and $\Gamma, x{:}\tau' \vdash e_1 : \tau_1$ :

$$\dfrac{\Gamma, x{:}\tau' \vdash e_0 : \tau_1 \to \tau \quad \Gamma, x{:}\tau' \vdash e_1 : \tau_1}{\Gamma, x{:}\tau' \vdash e : \tau}$$

  Therefore we can use the induction hypothesis to obtain $\Gamma \vdash e_0\{v/x\} : \tau_1 \to \tau$ and $\Gamma \vdash e_1\{v/x\} : \tau_1$. This therefore implies $\Gamma \vdash e\{v/x\} : \tau$ via the application rule.

- Case 4: a lambda abstraction

  - Case 4a: $e \equiv \lambda x{:}\tau''\ e_2$.
    Then $e\{v/x\} = e$ and therefore the result holds trivially.

– Case 4b: $e \equiv \lambda y{:}\tau''\ e_2$.

  Note that $y \notin FV[v]$, so $e\{v/x\} = \lambda y : \tau''.\, e_2\{v/x\}$ and the typing of $e$ looks like the following, where $\tau = \tau'' \to \tau_2$ :

$$\frac{\Gamma, x{:}\tau', y{:}\tau'' \vdash e_2 : \tau_2}{\Gamma, x{:}\tau', \vdash (\lambda y{:}\tau''.\, e_2) : \tau'' \to \tau_2}$$

  Use [exchange] and [app] to derive $\Gamma, y{:}\tau'', x{:}\tau' \vdash e_2 : \tau_2$ and then use the inductive hypothesis to obtain $\Gamma, y{:}\tau'' \vdash e_2\{v/x\} : \tau_2$. By [abs], therefore, $\Gamma \vdash (\lambda y{:}\tau''.\, e_2\{v/x\}) : \tau$ and we are done.

The next lecture covers the Progress lemma.