In lectures 20 and 21, we became acquainted with axiomatic semantics. We demonstrated that, given a precondition and a program known to terminate, we could verify that a given postcondition holds upon termination. Thus, axiomatic semantics would serve as a powerful technique for verifying program correctness. However, for many programming tasks, it could also prove too unwieldy. In order for the verifier to do its job, the programmer would need to precisely state the preconditions and postconditions for the program in addition to writing the program itself. This would require much additional work on the part of the programmer, who, as a general rule, prefers writing code to writing down axioms. So, what we would like is a more general-purpose, lightweight tool for verifying program correctness.

Type-checking is such a technique. While, unlike axiomatic semantics, type-checking usually cannot determine if a program will produce the correct output, it provides a mechanism to test whether or not a program will get stuck. That is, it can show that a type-correct program will never reach a non-final configuration in its operational semantics from where it cannot proceed to some other configuration. This is a very weak notion of program correctness, but it turns out to be very useful in practice.

We have already seen some typed languages in class this semester. For example, ML and the meta-language used in class are both typed. Today, we introduce a typed variant of lambda calculus, show how to construct operational and denotational semantics for this language, and discover some of its interesting properties.

1 Syntax

A typed lambda calculus (TLC) program is an expression containing no free variables. The syntax is virtually identical to that of untyped lambda calculus, with the exception of λ-terms. Since lambda abstraction defines a function expecting an argument, λ-terms in TLC programs should expect arguments of a certain type. In addition, TLC will allow a kind of expressions, corresponding to base values, such as integers, booleans, and unit values. Finally, TLC will define set of base types, corresponding to the base values. So, the complete syntax is given below:

$$\begin{array}{rcl} e & ::= & x \mid e_1\ e_2 \mid \lambda x:\tau.e \mid b \\ b & ::= & 0 \mid 1 \mid 2 \mid \ldots \mid \#\mathsf{t} \mid \#\mathsf{f} \mid \#\mathsf{u} \\ \tau & ::= & B \mid \tau_1 \rightarrow \tau_2 \\ B & ::= & \mathsf{int} \mid \mathsf{bool} \mid \mathsf{unit} \end{array}$$

The key difference between a the typed and untyped Lambda calculus is that **every TLC expression has an associated type**. For example, the expression 1 has the type int, which we can write as 1:int. Likewise, the function $TRUE_{int} = \lambda x : \mathsf{int}.\lambda y : \mathsf{int}.x$ has the type $\mathsf{int} \rightarrow (\mathsf{int} \rightarrow \mathsf{int})$. Since the $\rightarrow$ operator is right-associative, we can write

$$TRUE_{int} = (\lambda x : \mathsf{int}.\lambda y : \mathsf{int}.x) : \mathsf{int} \rightarrow \mathsf{int} \rightarrow \mathsf{int}$$

2 Small-Step Operational Semantics and Type Correctness

The small-step operational semantics in TLC are no different from those in untyped λ-calculus. The presence of types does not alter the evaluation rules for expressions, but merely limits on the kinds of expressions that may be evaluated. Below we give the evaluation context and small step operational semantics for TLC.

$$\begin{array}{rcl} C & ::= & C\ e \mid v\ C \mid [\cdot] \\ (\lambda x:\tau.e_1)e_2 & \rightarrow & e_1\{e_2/x\} \end{array}$$

Now, we are ready to revisit the concept of type correctness that we touched upon in the beginning of the lecture. If a program is type-correct then it cannot become stuck at any point during its execution. Thus, if a type-correct program e evaluates to some expression e' such that e' is not a base value or a λ-term, then e' must itself evaluate to some other expression e''. Formally,

$$\vdash e : \tau \Rightarrow (e \rightarrow^* e' \Rightarrow (e' \in Value \vee \exists e''.e' \rightarrow e''))$$

Above, the notation $\vdash e : \tau$ means *e is correct with respect to some type* τ. This notation will be used throughout our encounter with type-correctness.

By now, it is natural to inquire about what a type-incorrect TLC program would look like, and how it may get stuck. In answer to this question, recall our function definition for $TRUE_{int}$ above, and consider the following additional definition:

$$IF_{int} = \lambda t : \mathsf{int} \rightarrow \mathsf{int} \rightarrow \mathsf{int}.\lambda a : \mathsf{int}.\lambda b : \mathsf{int}\ t\ a\ b$$

Clearly, $IF_{int}(TRUE_{int}\ 2\ 3)$ will evaluate to 2. However, consider the expression $IF_{int}(\#\mathsf{t}\ 2\ 3) \rightarrow ((\#\mathsf{t}\ 2)\ 3)$. The expression $(\#\mathsf{t}\ 2)$ is meaningless, since $\#\mathsf{t}$ is not a function term. Therefore, the program gets stuck at this point.

3 Static Semantics and Type Checking

In order to analyze programs written in typed languages, we introduce a new kind of semantics called of the *static semantics*. Static semantics is a set of inference rules that defines the relationship between expressions and types of a language. In a moment, we will give the static semantics of typed lambda calculus, but prior to this, we must introduce the notion of *type context*.

A type context $\Gamma : \mathsf{Var} \rightarrow \mathsf{Type}$ is a map from variables to types. We can view it as a function that takes a variable and returns the variable's type. Equivalently, we can view Γ a set of pairs $x : \tau$, such that $x \in \mathsf{Var}$ and $\tau \in \mathsf{Type}$. We will hold to the first view, but from this point on, will use the traditional notation on the right-hand side of the equivalences below:

$$\begin{aligned} \Gamma(x) = \tau &\quad \Leftrightarrow \quad x : \tau \in \Gamma \\ \Gamma[x \mapsto \tau] &\quad \Leftrightarrow \quad \Gamma, x : \tau \end{aligned}$$

We are now ready to give the static semantics for typed Lambda calculus. We write $\Gamma \vdash e : \tau$ to signify that the expression e is correct with respect to type τ within the type context Γ. Below, we give the inference rules for well-typed TLC programs:

$$\frac{}{\Gamma \vdash n : \mathsf{int}} \qquad \frac{}{\Gamma \vdash \#\mathsf{t} : \mathsf{bool}} \qquad \frac{}{\Gamma \vdash \#\mathsf{u} : \mathsf{unit}}$$

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau \rightarrow \tau' \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\ e_2 : \tau'} \qquad \frac{(\Gamma, x : \tau) \vdash e : \tau'}{\Gamma \vdash (\lambda x : \tau.\ e) : \tau \rightarrow \tau'}$$

Let us explain the above inference rules in more detail. If e is an expression consisting of a single variable x, then $\Gamma \vdash e : \tau$ necessitates that x have type τ in the context Γ. If e is an application of e_1 to e_2 that has the type τ' , then e_1 must be a function from τ to τ', and e_2 must have the type τ. Finally, if the lambda-abstraction $\lambda(x : \tau.e)$ is a function of type $\tau \rightarrow \tau'$, then e must have the type τ' within the updated type context $(\Gamma, x : \tau)$, which we modify in order to account for the possibility that the variable x of type τ may appear among the free variables of e.

To type-check a TLC program, we can attempt to construct its proof tree. For example, consider the program $(\lambda x : \mathsf{int}.x)\ 2$, which evaluates to 2:int. We can construct a proof tree for this program as follows:

$$\frac{\dfrac{\dfrac{x : \mathsf{int} \in (\Gamma, x : \mathsf{int})}{(\Gamma, x : \mathsf{int}) \vdash x : \mathsf{int}}}{\Gamma \vdash (\lambda x : \mathsf{int}.\ x) : \mathsf{int} \rightarrow \mathsf{int}} \quad \Gamma \vdash 2 : \mathsf{int}}{\Gamma \vdash (\lambda x : \mathsf{int}.\ x)\ 2 : \mathsf{int}}$$

The above is a valid proof tree for our program. An automated type checker effectively constructs proof trees like this one in order to test whether a program is type-correct.

4 Expressive Power of Typed Lambda Calculus

By now you may be wondering if we have lost any expressive power of Lambda calculus by introducing types. The answer to this question is a resounding yes. First of all, we have lost generic function composition. We can no longer compose any two arbitrary functions, since they may have mismatching types. The IF_{int} function above is a good example of this.

Second, and perhaps more importantly, we have lost the ability to write down infinite loops. To convince ourselves of that, recall that an infinite loop in lambda calculus is defined as

$$LOOP = (\lambda x(xx))(\lambda x(xx))$$

Let us attempt to construct a static semantics proof tree for the TLC expression $(\lambda x : \tau.(x\ x))$:

$$\frac{\dfrac{(\Gamma, x:\tau) \vdash x : \tau \to \tau' \quad (\Gamma, x:\tau) \vdash x : \tau}{(\Gamma, x:\tau) \vdash (x\ x) : \tau'}}{\Gamma \vdash (\lambda x : \tau.\ (x\ x)) : \tau \to \tau'}$$

From the above, we see that $x : \tau \to \tau'$ and $x : \tau$. Therefore, we conclude that $\tau = \tau \to \tau'$. However, this not possible, since none of the types in the TLC type system are recursive. Therefore, we conclude that the expression $(\lambda x : \tau.(x\ x))$ is not type-correct, and hence we cannot write down infinite loops. In fact, a little later in the lecture, we will see that we cannot write down any non-terminating program.

5 Denotational Semantics

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 τ, and returns the domain associated with that type. For our type system, we define the function $\mathcal{T}$ below:

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

Note that $\mathcal{T}$ returns an entire domain corresponding to a type, not just an element of a domain. So, if a program e is correct with respect to type τ, then denotation of e is an element of $\mathcal{T}[\![\tau]\!]$. Formally,

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

Since each variable has a corresponding type, we need to establish a relation between the variable environment and the type context. We say that the variable environment ρ satisfies a type context Γ, written as $\rho \models \Gamma$, if for every variable-type pair $x : \tau$ in Γ, $\rho(x)$ is in the domain corresponding to τ. That is,

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

Finally, we need to modify the notation we use for the meaning function $\mathcal{C}$ to account for the presence of types. Instead of writing $\mathcal{C}[\![e]\!]\rho$, we will use the notation $\mathcal{C}[\![\Gamma \vdash e : \tau]\!]\rho$, where e has the type derivation τ in the context Γ. In other words, the semantic function $\mathcal{C}$ maps *type derivations* to meanings. We will not write the entire type derivation inside the semantic brackets, but it is implied. We expect that our denotational model satisfies the following condition:

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

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

$$\begin{aligned}
\mathcal{C}[\![\Gamma \vdash n : \mathsf{int}]\!]\rho &= n \\
\mathcal{C}[\![\Gamma \vdash \#\mathsf{t} : \mathsf{bool}]\!]\rho &= \mathsf{true} \\
\mathcal{C}[\![\Gamma \vdash \#\mathsf{u} : \mathsf{unit}]\!]\rho &= \#\mathsf{u} \\
\mathcal{C}[\![\Gamma \vdash x : \tau]\!]\rho &= \rho(x) \\
\mathcal{C}[\![\Gamma \vdash (e_1\ e_2) : \tau']\!]\rho &= (\mathcal{C}[\![\Gamma \vdash e_1 : \tau \to \tau']\!]\rho)\ (\mathcal{C}[\![\Gamma \vdash e_2 : \tau]\!]\rho) \\
\mathcal{C}[\![\Gamma \vdash (\lambda x : \tau.\ e : \tau \to \tau')]\!]\rho &= \lambda v \in \mathcal{T}[\![\tau]\!].\ \mathcal{C}[\![(\Gamma, x : \tau) \vdash\ e : \tau']\!]\ \rho[x \mapsto v]
\end{aligned}$$

Note that $\bot$ does not appear anywhere within the denotational semantics for typed lambda calculus. Therefore, we conclude that all typed lambda calculus programs terminate.