Scribe: Leonid Meyerguz, Andrew Myers Lecturer: Andrew Myers

Type checking is a lightweight technique for proving simple properties of programs. Unlike theorem-proving techniques based on axiomatic semantics, type checking usually cannot determine if a program will produce the correct output; instead, it is a way to test whether a program is *well-formed*, with the idea that a well-formed program satisfies certain simple properties. The traditional application of type checking is to show that a program cannot get stuck; that is, that a type-correct program will never reach a non-final configuration in its operational semantics from which its behavior is undefined. This is a very weak notion of program correctness, but it turns out to be very useful in practice for catching bugs.

We have already seen some typed languages in class this semester. For example, ML and the meta-language used in class are both typed. We now 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 ($\lambda^{\rightarrow}$) 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 $\lambda^{\rightarrow}$ programs should expect arguments of a certain type. In addition, $\lambda^{\rightarrow}$ will allow a kind of expressions, corresponding to base values, such as integers, booleans, and unit values. Finally, $\lambda^{\rightarrow}$ will define set of base types, corresponding to the base values. So, the complete syntax is given below:

$$\begin{array}{rcl} e & ::= & x \mid e_1\ e_2 \mid \lambda x{:}\tau.\, e \mid b \\ b & ::= & 0 \mid 1 \mid 2 \mid \ldots \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{unit} \\ \tau \in \mathit{Type} & ::= & B \mid \tau_1 \rightarrow \tau_2 \\ B & := & \mathsf{int} \mid \mathsf{bool} \mid 1 \\ v \in \mathit{Value} & ::= & b \mid \lambda x{:}\tau.\, e \end{array}$$

The key difference between a the typed and untyped Lambda calculus is that every $\lambda^{\rightarrow}$ expression has an associated type. For example, the expression 1 has the type int, which we can write as 1: int. The *type* 1 has nothing to do with integers; it is the type of the single unit value unit. Likewise, the function $TRUE_{\mathrm{int}} = \lambda x : \mathsf{int}.\lambda y : \mathsf{int}.x$ has the type $\mathsf{int} \rightarrow (\mathsf{int} \rightarrow \mathsf{int})$. Since the $\rightarrow$ operator is right-associative, we can write

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

2 Small-Step Operational Semantics and Type Correctness

The small-step operational semantics in $\lambda^{\rightarrow}$ 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 $\lambda^{\rightarrow}$.

$$C ::= C\ e \mid v\ C \mid [\cdot]$$

$$C[(\lambda x{:}\tau.\, e)v] \longrightarrow C[e\{v/x\}]$$

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 well-formed (well-typed) 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,

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

We use the notation $\vdash e : \tau$ to mean that e is well-typed with the type τ. This assertion is also called a *typing* for e.

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

$$\mathit{IF}_{\text{int}} = \lambda \mathsf{t} : \mathsf{int} \rightarrow \mathsf{int} \rightarrow \mathsf{int}.\, \lambda a : \mathsf{int}.\, \lambda \mathsf{b} : \mathsf{int}.\, t\ a\ b$$

Clearly, $\mathit{IF}_{\text{int}}(\mathit{TRUE}_{\text{int}}\ 2\ 3)$ will evaluate to 2. However, consider the expression $\mathit{IF}_{\text{int}}(\mathsf{true}\ 2\ 3) \rightarrow ((\mathsf{true}\ 2)\ 3)$. The expression $(\mathsf{true}\ 2)$ is meaningless, since true 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 *static semantics*. The name is a bit misleading – it's not really a semantics for the language, but rather a set of rules that define which programs are legal, usually expressed as 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 *typing context*.

A typing context Γ is an environment that maps variables to types. We can view it as a partial function that takes a variable and returns the variable's type: $\Gamma \in \mathit{Var} \rightharpoonup \mathit{Type}$. The notation $\mathit{dom}(\Gamma)$ is used to refer to the finite subset of the domain Var on which Γ is defined. The typing context $\emptyset$ is the empty typing context where $\mathit{dom}(\emptyset)$ is empty.

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 typing context Γ. The assertion $\vdash e : \tau$ we define as $\emptyset \vdash e : \tau$. Below, we give the inference rules for well-typed $\lambda^{\rightarrow}$ programs:

$$\frac{}{\Gamma \vdash n : \mathsf{int}} \qquad \frac{}{\Gamma \vdash \mathsf{true} : \mathsf{bool}} \qquad \frac{}{\Gamma \vdash \mathsf{unit} : 1}$$

$$\frac{}{\Gamma[x \mapsto \tau] \vdash x : \tau} \qquad \frac{\Gamma \vdash e_0 : \tau \rightarrow \tau' \quad \Gamma \vdash e_1 : \tau}{\Gamma \vdash e_0\ e_1 : \tau'} \qquad \frac{\Gamma[x \mapsto \tau] \vdash e : \tau'}{\Gamma \vdash (\lambda x : \tau.\ e) : \tau \rightarrow \tau'}$$

Let us explain these inference rules in more detail. If e is an expression consisting of a single variable x, then $\Gamma \vdash e : \tau$ requires that x have type τ in the typing context; that is, we must be able to find a context Γ such that the current typing context can be described as $\Gamma[x \mapsto \tau]$. (Often, as in Pierce's book, this is written as $\Gamma, x : \tau$.) If e is an application of e_0 to e_1 that has the type τ' , then e_0 must be a function from τ to τ', and e_1 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 typing context ($\Gamma[x \mapsto \tau]$, which we modify in order to account for the possibility that the variable x of type τ may legally appear among the free variables of e.

To type-check a $\lambda^{\rightarrow}$ 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:

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

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 loops. To convince ourselves of that, recall the term Ω that we defined as

$$\Omega = (\lambda x\ (x\ x))\ (\lambda x\ (x\ x))$$

Let us attempt to construct a derivation of a typing for the $\lambda^{\rightarrow}$ expression $(\lambda x{:}\tau.\, x\ x)$:

$$\dfrac{\dfrac{\Gamma[x \mapsto \tau] \vdash x : \tau \rightarrow \tau' \quad \Gamma[x \mapsto \tau] \vdash x : \tau}{\Gamma[x \mapsto \tau] \vdash (x\ x) : \tau'}}{\Gamma \vdash (\lambda x{:}\tau.\, x\ x) : \tau \rightarrow \tau'}$$

From the above, we see that $x : \tau \rightarrow \tau'$ and $x : \tau$. Therefore, we conclude that $\tau = \tau \rightarrow \tau'$. It is not possible to write down any finite type expression that satisfies this equation and therefore, we conclude that the expression $(\lambda x : \tau.(x\ x))$ cannot be typed. In fact, a little later in the lecture, we will see that we cannot write down *any* nonterminating program in $\lambda^{\rightarrow}$, at least according to a denotational model of what programs mean. This will turn out be true from an operational perspective as well. Fortunately, as we will see in later lectures, we can extend our type system to allow nonterminating programs.