CS611 Lecture 28 | Typed Lambda Calculus | 10/23/00, 11/5/01
Scribe: Leonid Meyerguz | Lecturer: Andrew Myers

In lectures 22 and 23, we became acquainted with axiomatic semantics. Given a precondition, a program known to terminate, and an oracle that generates proofs of formulas over integers, we could verify that a given postcondition holds upon termination. Thus, axiomatic semantics can serve as a powerful technique for verifying program correctness. However, for many programming tasks, it is too unwieldy. The program verifiers is a theorem prover; in order for it to do its job, the programmer formally states preconditions, postconditions, and invariants to help the theorem prover. This requires more additional work on the part of the programmer, and experience suggests that getting programmers to do this work is difficult. We would like a lighter-weight tool for checking that programs do not contain bugs.

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 which its behavior is undefined. 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 ($\lambda^{\rightarrow}$) 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:

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

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 $1\!:\ \mathsf{int}$. The *type* 1 has nothing to do with integers; it is the type of the single unit value $\#\mathsf{u}$. Likewise, the function $\mathit{TRUE}_{\text{int}} = \lambda x : \mathsf{int}.\lambda y : \mathsf{int}.x$ has the type $\mathsf{int} \rightarrow (\mathsf{int} \rightarrow \mathsf{int})$. Since the $\rightarrow$ operator is right-associative, we can write

$$\mathit{TRUE}_{\text{int}} = (\lambda x : \mathsf{int}.\lambda y : \mathsf{int}.x) : \mathsf{int} \rightarrow \mathsf{int} \rightarrow \mathsf{int}$$

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 $\lambda^{\rightarrow}$.

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

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

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 $\mathit{IF}_{\text{int}}(\#\mathsf{t}\ 2\ 3) \rightarrow ((\#\mathsf{t}\ 2)\ 3)$. 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 *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 subset of the domain Var on which Γ is defined. $\emptyset$ represents the typing context where $\mathit{dom}(\emptyset)$ is the empty set.

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:

$$\frac{}{\Gamma \vdash n : \mathsf{int}} \qquad \frac{}{\Gamma \vdash \#\mathsf{t} : \mathsf{bool}} \qquad \frac{}{\Gamma \vdash \#\mathsf{u} : 1}$$

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

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 $\Gamma[x \mapsto \tau]$. 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 $(\mathit{Gamma}[x \mapsto \tau]$, 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:

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

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 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.

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 this type system, we can define $\mathcal{T}$ in a straightforward way:

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

Note that $\mathcal{T}$ returns an entire domain corresponding to a type, not just an element of a domain. For the semantics of $\lambda^{\rightarrow}$, the domains will not need to have any ordering properties: they are just sets. So we have $\mathcal{T}[\![\cdot]\!] \in \mathit{Type} \rightarrow \mathit{Set}$. If a program e can be given type τ, 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 ρ we use, based on the typing context Γ. We say that the environment ρ satisfies a typing context Γ, written as $\rho \models \Gamma$, if for every variable-type mapping $x \mapsto \tau$ in Γ, $\rho(x)$ is in the domain corresponding to τ. That is,

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

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 τ in the context Γ. We will not write the entire typing derivation inside the semantic brackets, but it is implied. Therefore the expression e must be well-formed.

We expect that our denotational model satisfies the following soundness condition for all rho, Γ, e, x, τ:

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

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

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

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 $\perp$ 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.