1 Introduction

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 desirable properties. The traditional application of type checking is to show that a well-formed 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 weak notion of program correctness, but nevertheless very useful in practice for catching bugs.

Type systems are a powerful technique. In the past couple of decades, researchers have discovered how to use type systems for a variety of different verification tasks.

2 $\lambda^{\rightarrow}$

We have already seen some typed languages in class this semester. For example, OCaml and the metalanguage used in class for denotational semantics are both typed.

To explore the idea of type checking itself, we introduce $\lambda^{\rightarrow}$, a typed variant of the λ-calculus in which we assign types to certain λ-terms according to some typing rules. A λ-term is considered to be well-formed if a type can be derived for it using the rules. We will give operational and denotational semantics for this language. Along the way, we will discover some interesting properties that give insight about typing in more complex languages.

3 Syntax

The syntax of $\lambda^{\rightarrow}$ is similar to that of untyped λ-calculus, with some notable differences. There are two kinds of inductively-defined expressions, *terms* and *types*:

$$\begin{array}{lcll} \text{terms} & e & ::= & n \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{null} \mid x \mid e_1\, e_2 \mid \lambda x : \tau .\, e \\ \text{types} & \tau & ::= & \mathsf{int} \mid \mathsf{bool} \mid \mathsf{unit} \mid \tau_1 \to \tau_2 \end{array}$$

One difference is that the natural numbers, Boolean constants, and null are taken to be primitive symbols and not defined as λ-terms. For this reason, we no longer need to distinguish between the syntactic objects $\overline{n}$ and true and their semantic counterparts n and *true*, so we might as well just identify them. Another difference is that a λ-abstraction explicitly mentions the type of its argument. Thus $\lambda x : \tau .\, e$ represents a function that takes an input value of type τ and evaluates e.

A *value* is either a number, a Boolean constant, null, or a closed abstraction $\lambda x : \tau .\, e$. The set of values is denoted *Val*. The set of types is denoted *Type*.

A *type* represents a collection of related values. The types int, bool, and unit represent integers, Booleans, and null, respectively. The type $\tau_1 \to \tau_2$ represents functions that take inputs of type τ_1 and produce outputs of type τ_2.

There are some *typing rules*, given below, that can be used to associate a type with a term. The rules are used to derive *type judgments* of the form $\Gamma \vdash e : \tau$, which means intuitively that under the assumptions Γ, the term e has type τ. The assumptions Γ specify types for the free variables of e. We write $\vdash e : \tau$ when e is a closed term and Γ is empty.

For example, every number has type int, thus $\vdash 3 : \mathsf{int}$. The type unit is the type of the value null, and nothing else has this type. The function $\mathsf{true}_{\text{int}} = \lambda x : \mathsf{int}.\, \lambda y : \mathsf{int}.\, x$ has the type $\mathsf{int} \to \mathsf{int} \to \mathsf{int}$. By convention, the type constructor $\to$ associates to the right, so $\mathsf{int} \to \mathsf{int} \to \mathsf{int}$ is the same as $\mathsf{int} \to (\mathsf{int} \to \mathsf{int})$. Thus we can write

$$\mathsf{true}_{\text{int}} \quad \triangleq \quad (\lambda x : \mathsf{int}.\, \lambda y : \mathsf{int}.\, x) : \mathsf{int} \to \mathsf{int} \to \mathsf{int}.$$

Not all λ-terms can be typed, for instance $\lambda x : \mathsf{int}.\, xx$ or $\mathsf{true}\, 3$. These expressions are considered ill-formed and nonsensical.

Right now, we cannot do anything interesting with integers or Booleans, because we do not have any operations on them. Later on we will be adding other typed constants such as $\mathsf{plus} : \mathsf{int} \to \mathsf{int} \to \mathsf{int}$ and $\mathsf{equal} : \mathsf{int} \to \mathsf{int} \to \mathsf{bool}$, but for now they are just there to set the stage.

4 Small-Step Operational Semantics and Type Correctness

The small-step CBV operational semantics of $\lambda^{\to}$ is the same as that of the untyped λ-calculus. The presence of types does not alter the evaluation rules for expressions.

$$E \ ::= \ E\, e \ \mid \ v\, E \ \mid \ [\cdot] \qquad\qquad (\lambda x : \tau.\, e)\, v \ \to \ e\{v/x\}$$

We will eventually show that these reduction rules preserve typing in the sense that if e has type τ and $e \xrightarrow{*} e'$, then e' also has type τ. Thus "well-typedness" of programs is invariant under the evaluation rules of the language. This property is known as *type preservation* or *subject reduction*. Another important property is *progress*, which says that a well-typed program is never stuck; that is, it is either a value or a further transition is possible. These two properties together imply that no well-typed term can ever become stuck. Thus the typing rules can be used in place of runtime type checking to ensure strong typing.

It is natural to ask what a type-incorrect term might look like and how it could get stuck. Recall our function definition for $\mathsf{true}_{\text{int}}$ above, and consider the following additional definition:

$$\mathsf{if}_{\text{int}} \quad \triangleq \quad \lambda t : \mathsf{int} \to \mathsf{int} \to \mathsf{int}.\, \lambda a : \mathsf{int}.\, \lambda b : \mathsf{int}.\, t\, a\, b.$$

Clearly, $\mathsf{if}_{\text{int}}\, \mathsf{true}_{\text{int}}\, 2\, 3$ evaluates to 2. However, $\mathsf{if}_{\text{int}}\, \mathsf{true}\, 2\, 3 \to \mathsf{true}\, 2\, 3$, and this expression is meaningless, since true is not a function and cannot be applied to anything. Therefore, the program would be stuck at this point.

5 Typing Rules

The typing rules will determine which terms are well-formed $\lambda^{\to}$ programs. They are a set of rules that allow the derivation of *type judgments* of the form $\Gamma \vdash e : \tau$. Here Γ is a *type environment*, a partial map from variables to types used to determine the types of the free variables in e. The domain of Γ as a partial function $\mathit{Var} \rightharpoonup \mathit{Type}$ is denoted $\mathsf{dom}\, \Gamma$.

The environment $\Gamma[\tau/x]$ is obtained by rebinding x to τ (or creating the binding anew if $x \notin \mathsf{dom}\, \Gamma$):

$$\Gamma[\tau/x](y) \quad \triangleq \quad \begin{cases} \Gamma(y), & \text{if } y \neq x \text{ and } y \in \mathsf{dom}\, \Gamma, \\ \tau, & \text{if } y = x, \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

The notation $\Gamma,\, x:\tau$ is synonymous with $\Gamma[\tau/x]$. The former is standard notation in the literature. Also, one often sees $x:\tau \in \Gamma$, which means just $\Gamma(x) = \tau$.

We also write $\Gamma \vdash e:\tau$ as a metaexpression to mean that the type judgment $\Gamma \vdash e:\tau$ is derivable from the typing rules. The environment $\varnothing$ is the empty environment, and the judgment $\vdash e:\tau$ is short for $\varnothing \vdash e:\tau$.

Here are the typing rules:

$$\Gamma \vdash n : \mathsf{int} \qquad \Gamma \vdash \mathsf{true} : \mathsf{bool} \qquad \Gamma \vdash \mathsf{false} : \mathsf{bool} \qquad \Gamma \vdash \mathsf{null} : \mathsf{unit} \qquad \Gamma,\, x:\tau \vdash x:\tau$$

$$\frac{\Gamma \vdash e_0 : \sigma \to \tau \quad \Gamma \vdash e_1 : \sigma}{\Gamma \vdash e_0\, e_1 : \tau} \qquad \frac{\Gamma,\, x:\sigma \vdash e:\tau}{\Gamma \vdash (\lambda x:\sigma.\, e) : \sigma \to \tau}$$

Let us explain these rules in more detail.

- The first four rules just say that all the base values have their corresponding base types.
- For a variable x, $\Gamma \vdash x:\tau$ holds if the binding $x:\tau$ appears in the type environment Γ; that is, if $\Gamma(x) = \tau$.
- An application expression $e_0\, e_1$ represents the result of applying the function represented by e_0 to the argument represented by e_1. For this to have type τ, e_0 must be a function of type $\sigma \to \tau$ for some σ, and its argument e_1 must have type σ. This is captured in the typing rule for $e_0\, e_1$.
- Finally, a λ-abstraction $\lambda x:\sigma.\, e$ is supposed to represent a function. The type of the input should match the annotation in the term, thus the type of the function must be $\sigma \to \tau$ for some τ. The type τ of the result is the type of the body under the extra type assumption $x:\sigma$. This idea is captured in the typing rule for λ-abstractions.

Every well-typed $\lambda^{\to}$ term has a proof tree consisting of applications of the typing rules to derive a type for the term. We can type-check a term by constructing this proof tree. For example, consider the program $(\lambda x:\mathsf{int}.\, \lambda y:\mathsf{bool}.\, x)\; 2\; \mathsf{true}$, which evaluates to 2. Since $\vdash 2:\mathsf{int}$, we expect $\vdash ((\lambda x:\mathsf{int}.\, \lambda y:\mathsf{bool}.\, x)\; 2\; \mathsf{true}):\mathsf{int}$ as well. Here is a proof of that fact:

$$\dfrac{\dfrac{\dfrac{\dfrac{x:\mathsf{int},\, y:\mathsf{bool} \vdash x:\mathsf{int}}{x:\mathsf{int} \vdash (\lambda y:\mathsf{bool}.\, x):\mathsf{bool} \to \mathsf{int}}}{\vdash (\lambda x:\mathsf{int}.\, \lambda y:\mathsf{bool}.\, x):\mathsf{int} \to \mathsf{bool} \to \mathsf{int}} \quad \vdash 2:\mathsf{int}}{\vdash ((\lambda x:\mathsf{int}.\, \lambda y:\mathsf{bool}.\, x)\; 2):\mathsf{bool} \to \mathsf{int}} \qquad \vdash \mathsf{true}:\mathsf{bool}}{\vdash ((\lambda x:\mathsf{int}.\, \lambda y:\mathsf{bool}.\, x)\; 2\; \mathsf{true}):\mathsf{int}}$$

An automated type checker can effectively construct proof trees like this in order to test whether a program is type-correct.

Note that types, if they exist, are unique. That is, if $\Gamma \vdash e:\tau$ and $\Gamma \vdash e:\tau'$, then $\tau = \tau'$. This can be proved easily by structural induction on e, using the fact that there is exactly one typing rule that applies in each case, depending on the form of e.

6 Expressive Power

By now you may be wondering if we have lost any of the expressive power of λ-calculus by introducing types. The answer to this question is a resounding *yes*. For example, we can no longer compose arbitrary functions, since they may have mismatched types.

More importantly, we have lost the ability to write loops. Recall the paradoxical combinator

$$\Omega \;\triangleq\; (\lambda x.\, xx)\,(\lambda x.\, xx).$$

Let us show that any attempt to derive a typing for the term $\lambda x\!:\!\sigma.\, xx$ must fail:

$$\dfrac{\dfrac{\Gamma,\, x\!:\!\sigma \vdash x : \sigma \to \tau \quad \Gamma,\, x\!:\!\sigma \vdash x : \sigma}{\Gamma,\, x\!:\!\sigma \vdash xx : \tau}}{\Gamma \vdash (\lambda x\!:\!\sigma.\, xx) : \sigma \to \tau}$$

We see that we must have both $\Gamma,\, x\!:\!\sigma \vdash x : \sigma \to \tau$ and $\Gamma,\, x\!:\!\sigma \vdash x : \sigma$. However, since types are unique, this is impossible; we cannot have $\sigma = \sigma \to \tau$, since no type expression can be a subexpression of itself. We conclude that the term $\lambda x\!:\!\sigma.\, xx$ cannot be typed.

In fact, we will see later that we cannot write down *any* nonterminating program in $\lambda^{\to}$. This will turn out be true from an operational perspective as well. In later lectures we will show how to extend the type system to allow loops and nonterminating programs.

7 Denotational Semantics

Before we can give the denotational semantics for $\lambda^{\to}$, we need to define a new meaning function $\mathcal{T}[\![\cdot]\!]$ that maps each type to a domain associated with that type. For this type system, the definition of $\mathcal{T}[\![\cdot]\!]$ is straightforward:

$$\mathcal{T}[\![\mathsf{int}]\!] \triangleq \mathbb{Z} \qquad \mathcal{T}[\![\mathsf{bool}]\!] \triangleq \mathbb{2} \qquad \mathcal{T}[\![\mathsf{unit}]\!] \triangleq \{\mathsf{null}\} \qquad \mathcal{T}[\![\sigma \to \tau]\!] \triangleq \mathcal{T}[\![\sigma]\!] \to \mathcal{T}[\![\tau]\!].$$

In the last equation, note that the $\to$ on the left-hand side is just a symbol in the language of types, a syntactic object, whereas the $\to$ on the right-hand side is a semantic object, namely the operator that constructs the space of functions between a given domain and range. For now, these domains need not have any ordering properties; they are just sets. So we have $\mathcal{T}[\![\cdot]\!] : \mathit{Type} \to \mathit{Set}$.

For any closed term e, if $\vdash e : \tau$, then we expect the denotation of e to be an element of $\mathcal{T}[\![\tau]\!]$. More generally, for a term e possibly containing free variables, if there is a type environment Γ and a value environment ρ such that ρ and Γ are defined on all the free variables of e and $\rho(x) \in \mathcal{T}[\![\Gamma(x)]\!]$ for all $x \in \mathit{FV}(e)$, and if $\Gamma \vdash e : \tau$, then we expect the denotation of e in environment ρ to be an element of $\mathcal{T}[\![\tau]\!]$.

We say that the value environment ρ *satisfies* a type environment Γ and write $\rho \vDash \Gamma$ if $\mathsf{dom}\,\Gamma \subseteq \mathsf{dom}\,\rho$ and for every $x \in \mathsf{dom}\,\Gamma$, $\rho(x) \in \mathcal{T}[\![\Gamma(x)]\!]$.

We are now ready to give the denotational semantics for typed λ-terms. We only define the meaning function for well-typed expressions. The following function is defined only for e, Γ, and ρ such that $\rho \vDash \Gamma$ and e is well-typed under Γ; that is, $\Gamma \vdash e : \tau$ for some type τ.

$$\begin{aligned}
\mathcal{C}[\![n]\!]\,\Gamma\,\rho &\triangleq n \\
\mathcal{C}[\![\mathsf{true}]\!]\,\Gamma\,\rho &\triangleq \mathsf{true} \\
\mathcal{C}[\![\mathsf{false}]\!]\,\Gamma\,\rho &\triangleq \mathsf{false} \\
\mathcal{C}[\![\mathsf{null}]\!]\,\Gamma\,\rho &\triangleq \mathsf{null} \\
\mathcal{C}[\![x]\!]\,\Gamma\,\rho &\triangleq \rho(x), \quad x \in \mathsf{dom}\,\Gamma \\
\mathcal{C}[\![e_0\,e_1]\!]\,\Gamma\,\rho &\triangleq (\mathcal{C}[\![e_0]\!]\,\Gamma\,\rho)\,(\mathcal{C}[\![e_1]\!]\,\Gamma\,\rho) \\
\mathcal{C}[\![\lambda x\!:\!\tau.\, e]\!]\,\Gamma\,\rho &\triangleq \lambda v \in \mathcal{T}[\![\tau]\!]\,.\,\mathcal{C}[\![e]\!]\,\Gamma[\tau/x]\,\rho[v/x]
\end{aligned}$$

8 Soundness

It is possible to show that $\mathcal{C}[\![\cdot]\!]$ satisfies the following soundness condition: for all ρ, Γ, e, and τ,

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

The proof is by induction on the structure of e. The only interesting step is the case of a λ-term, which requires the observation

$$\rho \models \Gamma \;\wedge\; v \in \mathcal{T}[\![\tau]\!] \quad \Rightarrow \quad \rho[v/x] \models \Gamma[\tau/x].$$

Note that $\perp$ does not appear anywhere in this semantics. We did not need it because we never took a fixpoint. We can conclude that all well-typed λ-terms represent total functions.

Of course, we will also want to see that the operational semantics is adequate with respect to this model to ensure that our evaluation relation actually produces the values of the correct type that the denotational model advertises are there. For example, for a closed term e such that $\vdash e : \mathsf{int}$, the adequacy condition would say that

$$e \xrightarrow{*} n \quad \Leftrightarrow \quad \mathcal{C}[\![e]\!]\varnothing\varnothing = n.$$