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 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, ML 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, but adds types. To make our examples more interesting, we include a few primitive types: **int**, **bool**, **unit**:

primitive values	b	::=	$n \mid \mathbf{true} \mid \mathbf{false} \mid \mathbf{null}$
terms	e	::=	$b \mid x \mid e_1\, e_2 \mid \lambda x{:}\tau.\, e$
primitive types	B	::=	$\mathbf{int} \mid \mathbf{bool} \mid \mathbf{unit}$
types	τ	::=	$\tau_1 \rightarrow \tau_2 \mid B$

One difference is that the natural numbers, Boolean constants, and **null** are taken to be primitive symbols and not defined as λ-terms. Another difference is that a λ-abstraction explicitly mentions the type of its argument.

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**.

There is a set of *typing rules*, given below, that can be used to associate a type with a term. If a type τ can be derived for a term e according to the typing rules, we write $\vdash e : \tau$. This is called a *type judgement*.

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

$$TRUE_{\text{int}} \triangleq (\lambda x{:}\mathbf{int}.\, \lambda y{:}\mathbf{int}.\, x) : \mathbf{int} \rightarrow \mathbf{int} \rightarrow \mathbf{int}.$$

Not all λ-terms can be typed, for instance $(\lambda x{:}\mathbf{int}.\, x\ x)$ and $(\mathbf{true}\ 3)$. These expressions are not well-formed and are considered 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 $\mathbf{plus} : \mathbf{int} \rightarrow \mathbf{int} \rightarrow \mathbf{int}$ and $\mathbf{equal} : \mathbf{int} \rightarrow \mathbf{int} \rightarrow \mathbf{bool}$, 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 $\lambda^{\rightarrow}$ is the same as that of the untyped λ-calculus. The presence of types does not alter the evaluation rules for expressions.

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

We will eventually show that these reduction rules preserve well-formedness in the sense that if e has type τ and $e \xrightarrow{*} e'$, then e' also has type τ. Coupled with the observation that all well-typed terms are either values or can be further reduced, this says that no well-typed term can become stuck. Thus the typing rules can be used in place of run-time 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 $TRUE_{\text{int}}$ above, and consider the following additional definition:

$$IF_{\text{int}} \;\triangleq\; \lambda t{:}\mathbf{int} \rightarrow \mathbf{int} \rightarrow \mathbf{int}.\, \lambda a{:}\mathbf{int}.\, \lambda b{:}\mathbf{int}.\, t\ a\ b.$$

Clearly, $IF_{\text{int}}\ TRUE_{\text{int}}\ 2\ 3$ evaluates to 2. However, $IF_{\text{int}}\ \mathbf{true}\ 2\ 3 \rightarrow \mathbf{true}\ 2\ 3$, 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 $\lambda^{\rightarrow}$ programs. They are a set of rules that allow the derivation of *type judgements* 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 $\mathbf{Var} \rightharpoonup \mathbf{Type}$ is denoted $\mathrm{dom}(\Gamma)$.

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

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

The notation $\Gamma, x{:}\tau$ is synonymous with $\Gamma[x \mapsto \tau]$. 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 $\Gamma \vdash e : \tau$ to mean that the type judgement $\Gamma \vdash e : \tau$ is derivable from the typing rules. The environment $\varnothing$ is the empty environment, and the judgement $\vdash e : \tau$ is short for $\varnothing \vdash e : \tau$.

The typing rules are:

$$\Gamma \vdash n : \mathbf{int} \qquad \Gamma \vdash \mathbf{true} : \mathbf{bool} \qquad \Gamma \vdash \mathbf{false} : \mathbf{bool} \qquad \Gamma \vdash \mathbf{null} : \mathbf{unit} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

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

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 $e_0\, e_1$ 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 $\tau \rightarrow \tau'$ for some τ', and its argument e_1 must have type τ. This is captured in the typing rule for $e_0\, e_1$.

- Finally, a λ-abstraction $\lambda x:\tau.\, e$ 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 $\tau \to \tau'$ for some τ'. The type τ' of the result is the type of the body under the extra type assumption $x{:}\tau$. 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 $(\lambda x{:}\,\mathbf{int}.\,\lambda y{:}\,\mathbf{bool}.\,x)\ 2\ \mathbf{true}$, which evaluates to 2. Since $\vdash 2 : \mathbf{int}$, we expect $\vdash ((\lambda x{:}\,\mathbf{int}.\,\lambda y{:}\,\mathbf{bool}.\,x)\ 2\ \mathbf{true}) : \mathbf{int}$ as well. Here is a proof of that fact:

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

An automated type checker can effectively construct proof trees like this in order to test whether a program is type-correct. One key reason that this is possible is that the rules are *syntax-directed*: there is exactly one typing rule that applies to a given form of e. So there is no searching needed to construct the proof tree.

Note that for this type system, 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 for each expression form.

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

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

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

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

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, defining $\mathcal{T}[\![\cdot]\!]$ is straightforward, almost trivial:

$$\begin{aligned}
\mathcal{T}[\![\mathbf{int}]\!] &= \mathbb{Z} \\
\mathcal{T}[\![\mathbf{bool}]\!] &= \{\mathit{true}, \mathit{false}\} \\
\mathcal{T}[\![\mathbf{unit}]\!] &= \{*\} \\
\mathcal{T}[\![\tau \to \tau']\!] &= \mathcal{T}[\![\tau]\!] \to \mathcal{T}[\![\tau']\!].
\end{aligned}$$

In the last equation, note that the $\rightarrow$ on the left-hand side is just a symbol in the language of types, a syntactic object, whereas the $\rightarrow$ 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 $\mathcal{T}[\![\cdot]\!] : \mathbf{Type} \rightarrow \mathbf{Set}$.

For any closed term e, if $\vdash e : \tau$, 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 $x \in FV(e)$, 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 $\rho \models \Gamma$ if $\mathrm{dom}(\Gamma) \subseteq \mathrm{dom}(\rho)$ and for every $x \in \mathrm{dom}(\Gamma)$, $\rho(x) \in \mathcal{T}[\![\Gamma(x)]\!]$.

We only define the meaning function for well-typed expressions. This kind of semantics—where we do not assign any meaning to ill-formed terms—is known as a *Church-style semantics*. The alternative is *Curry-style semantics*, in which we give a meaning even to programs that are not well-typed. Our operational semantics, for example, talks about how to evaluate even ill-typed terms.

Therefore, we modify the the meaning function $\mathcal{C}$ to limit it to taking the meaning of 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 write $\mathcal{C}[\![\Gamma \vdash e : \tau]\!]\rho$, 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-typed. Further, when defining the meaning of an expression e, we can assume that the meaning of the expressions that appear in the premises of e's typing derivation is also well-defined.

$$
\begin{aligned}
\mathcal{C}[\![\Gamma \vdash n : \mathbf{int}]\!]\rho &= n \in \mathbb{Z} \\
\mathcal{C}[\![\Gamma \vdash \mathbf{true} : \mathbf{bool}]\!]\rho &= \mathit{true} \\
\mathcal{C}[\![\Gamma \vdash \mathbf{null} : \mathbf{unit}]\!]\rho &= * \\
\mathcal{C}[\![\Gamma \vdash x : \tau]\!]\rho &= \rho(x) \\
\mathcal{C}[\![\Gamma \vdash e_0\ e_1 : \tau']\!]\rho &= (\mathcal{C}[\![\Gamma \vdash e_0 : \tau \rightarrow \tau']\!]\rho)\ (\mathcal{C}[\![\Gamma \vdash e_1 : \tau]\!]\rho) \\
\mathcal{C}[\![\Gamma \vdash (\lambda x{:}\tau.\, e) : \tau \rightarrow \tau']\!]\rho &= \lambda v \in \mathcal{T}[\![\tau]\!].\, \mathcal{C}[\![\Gamma, x{:}\tau \vdash e : \tau']\!]\rho[x \mapsto v]]
\end{aligned}
$$

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:

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

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, according to this model, all simply-typed lambda calculus programs terminate. 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. We can express the idea that the two semantics agree at a type τ using the following adequacy conditions:

$$\vdash e : \tau \wedge e \longrightarrow^* v \implies \mathcal{C}[\![\vdash e : \tau]\!]\emptyset = \mathcal{C}[\![\vdash v : \tau]\!]\emptyset$$

$$\vdash e : \tau \wedge \mathcal{C}[\![\vdash e : \tau]\!]\emptyset = d \implies e \longrightarrow^* v \wedge \mathcal{C}[\![\vdash v : \tau]\!] = d$$

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