CMPSC 461: Programming Language Concepts
Note 3: Simply Typed Lambda Calculus

1 Introduction

Type checking is a lightweight technique for proving simple properties of programs. Type checking usually cannot determine if a program will produce the correct output or not. 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. In other words, a type-correct program will never reach a state from which the programs behavior is undefined (e.g., adding two floats by an integer addition). 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, such as verifying information flow security at compile time.

2 A simply typed lambda calculus

To understand how type system works, we consider a typed variant of the λ-calculus we have seen earlier in this class. In this language, 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. In this lecture, we will not formalize and prove the properties of this type system. But we will see the properties informally in the end.

2.1 Syntax

The language syntax is similar to that of untyped λ-calculus, with some notable differences. There are two kinds of inductively-defined expressions, terms and types. The definition of this language can be written in context-free grammar as follows. As standard, we use e_1 and e_2 to distinguish multiple uses of the same nonterminal e in the grammar.

terms	$e ::= n \mid \texttt{true} \mid \texttt{false} \mid x \mid e_1\ e_2 \mid \lambda x : \tau\ .\ e \mid e_1 + e_2 \mid e_1 \wedge e_2$
types	$\tau ::= \texttt{int} \mid \texttt{bool} \mid \tau_1 \to \tau_2$

One difference from the pure λ-calculus is that the natural numbers (n) and Boolean constants (`true` and `false`) are taken to be primitive symbols (just as in C and Scheme). This language also contains primitive operations: the plus operation ($+$) on natural numbers, as well as the and operation ($\wedge$) on Boolean values. Another difference is that a λ-abstraction explicitly mentions the type of its argument (symbol τ in the term $\lambda x : \tau\ .\ e$ is the type of parameter x). A type τ in this language is either a primitive type (`int`, `bool`), or a constructed function type. Here, $\tau_1 \to \tau_2$ represents a function from type τ_1 to type τ_2.

A *value* is either a number, a Boolean constant, or a closed λ-abstraction $\lambda x : \tau\ .\ e$. 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 $\vdash e : \tau$, read as "e has type τ". This metaexpression is called a *type judgment*. For example, every number has type `int`, thus $\vdash 3 : \texttt{int}$. Boolean value `true` has type `bool`, thus $\vdash \texttt{true} : \texttt{bool}$. A function $\lambda x : \texttt{int}\ .\ \lambda y : \texttt{int}\ .\ x$ has the type $\texttt{int} \to (\texttt{int} \to \texttt{int})$. The $\to$ constructor associates to the right, so $\texttt{int} \to (\texttt{int} \to \texttt{int})$ is the same as $\texttt{int} \to \texttt{int} \to \texttt{int}$. Thus we can write

$$\vdash (\lambda x : \texttt{int}\ .\ \lambda y : \texttt{int}\ .\ x) : \texttt{int} \to \texttt{int} \to \texttt{int}$$

Not all λ-terms can be typed, for instance $(\lambda x : int\ .\ x\ x)$ or $(\texttt{true}\ 3)$. These expressions are considered nonsensical. Intuitively, such terms are ill-formed. These terms should be rejected by a type system that ensures type-safety. Next, we will define a type system that ensures that any well-typed term will never get stuck. For example, both $(\lambda x : \texttt{int}.\ x\ x)$ and $(\texttt{true}\ 3)$ are rejected by the type system.

2.2 Typing rules

The typing rules will determine which terms are well-formed terms. They are a set of rules that allow the derivation of type judgments of the form $\Gamma \vdash e : \tau$, read as "e has type τ in context Γ".

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, written as $\texttt{dom}(\Gamma)$, is a subset of variables, whose types are defined in the type environment. The environment $\Gamma[\tau/x]$ is obtained by rebinding x to τ in Γ (or creating the binding anew if x is not in the domain of Γ):

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

We can also view type context Γ as a sequence of variables and their types. The notation $\Gamma, x : \tau$, which is equivalent to $\Gamma[\tau/x]$, follows this view. In this lecture, we also use $\Gamma, x : \tau$ to represent that τ is the most recent binding of x in the context. This notation is used frequently 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 $\emptyset$ is the empty environment, and the judgment $\vdash e : \tau$ is short for $\emptyset \vdash e : \tau$. The typing rules are:

$\Gamma \vdash n : \texttt{int}$ (T-NUM) $\quad \Gamma \vdash \texttt{true} : \texttt{bool}$ (T-TRUE) $\quad \Gamma \vdash \texttt{false} : \texttt{bool}$ (T-FALSE) $\quad \Gamma, x : \tau \vdash x : \tau$ (T-VAR)

$$\frac{\Gamma \vdash e_1 : \tau \to \tau' \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\ e_2 : \tau'}\ \text{(T-APP)} \qquad \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash (\lambda x : \tau\ .\ e) : \tau \to \tau'}\ \text{(T-ABS)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash (e_1 + e_2) : \texttt{int}}\ \text{(T-ADD)} \qquad \frac{\Gamma \vdash e_1 : \texttt{bool} \quad \Gamma \vdash e_2 : \texttt{bool}}{\Gamma \vdash (e_1 \wedge e_2) : \texttt{bool}}\ \text{(T-AND)}$$

The typing rules are presented as *inference rules*, which inductively define relations consisting of the derivation of types. Each typing rule has three parts: the possibly empty assumption (type judgments above the bar), the conclusion (the type judge below the bar), and a name for the typing rule (in parentheses). When the assumption is empty, we simply write the conclusion without the bar.

Now let's take a closer look at these typing rules:

- The first three rules just say that all the base values have their corresponding base types.
- (T-Var) For a variable x, x has a type τ if the binding $x : \tau$ appears as the most recent binding of x in the type environment.
- (T-App) An application expression $e_1\ e_2$ represents the result of applying the function represented by e_1 to the argument represented by e_2. For this to have type τ', e_1 must be a function of type $\tau \to \tau'$ for some τ, and its argument e_2 must have type τ.
- (T-Abs) 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 $x : \tau$.
- The last two rules say the operands of add (resp. and) must have type `int` (resp. `bool`), and the result has type `int` (resp. `bool`).

Every well-typed 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 term $\lambda x : \texttt{int} \,.\, (x+1)$, which takes a natural number x, and returns $x+1$, another natural number. So we expect $\vdash (\lambda x : \texttt{int} \,.\, (x+1)) : \texttt{int} \to \texttt{int}$. Here is a proof of that fact:

$$\dfrac{\dfrac{x : \texttt{int} \vdash x : \texttt{int} \qquad x : \texttt{int} \vdash 1 : \texttt{int}}{x : \texttt{int} \vdash (x+1) : \texttt{int}}}{\vdash (\lambda x : \texttt{int} \,.\, (x+1)) : \texttt{int} \to \texttt{int}}$$

Type checking starts from the bottom judgement. It has an empty typing environment (there is no global variable), and the goal of type checking is to derive the type on the right (initially unknown). To derive the type, at each step, we use the typing rule that matches the term being typed. Here, we first apply rule T-Abs to check function body $x + 1$ under environment $x : \texttt{int}$. The rule application gives us the second to last judgement in the proof tree, whose type after ':' is still unknown at at this moment. Next, we apply rule T-Add to type-check x and 1. Here, we can fill in the rightmost types, since axioms T-Var and T-Num apply, and we no longer need the assumptions to derive the correct type. With the complete assumption, we can finish the type of $x + 1$ using rule T-Add, and similarly, fill in the type in the last judgement to complete the proof tree. Deriving types using a proof tree typically involves this "bottom-up" and then "top-down" flavor. You can use the next proof trees as an excise.

Consider another term $(\lambda x : \texttt{int} \,.\, \lambda y : \texttt{bool} \,.\, x)\ 2\ \texttt{true}$, which evaluates to 2. Since $\vdash 2 : \texttt{int}$,we expect $\vdash ((\lambda x : \texttt{int} \,.\, \lambda y : \texttt{bool} \,.\, x)\ 2\ \texttt{true}) : \texttt{int}$ as well. Here is a proof of that fact:

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

An automated type checker can effectively construct proof trees like this in order to test whether a program is type-correct. A term *type-checks* if there is a proof tree for that term; otherwise, the term is rejected by the type system. Most type-systems are decidable, and in fact, efficient. The complexity of most type systems ensuring type safety (including the one in this lecture) is linear to program size.

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.

3 Type Safety

The type system enforces a very important property. Informally, we can state the property as follows: if a term e has type τ in context Γ, and e is executed in a memory state that is consistent with Γ (e.g., x has value 1 if $\Gamma(x) = \texttt{int}$), the final value of e must be a value of type τ. One implication is that since the type system checks that all operands to an operator (e.g., + and $\wedge$ in our language) have the proper type, an operator can never get a value with wrong type at run time. Hence, any well-formed program is type safe. However, the formal definition and proof of this property are beyond the scope of this lecture.