Class 15 - Type Checking and Type Inference

As we have seen in the prior classes, dealing with type coercions and checking for dynamic type errors complicate the operational semantics of a language and, in turn, the interpreter implementation. Some languages restrict the possible programs that can be executed to those that are guaranteed to not result in a dynamic type error. This restriction of programs is enforced with an analysis phase after parsing known as type checking. Such languages are called strongly, statically-typed. In this class, we will study a simple strongly statically-typed variant of JAKARTA SCRIPT.

Type Checking

The following expressions in our language from class 11 will result in a dynamic type error during evaluation:

\[
(3 + 4) (0)
\]
\[
1 \times 3 == \text{function}(x) 0
\]

The first expression will fail when we try to evaluate the call because the callee expression \((3 + 4)\) does not evaluate to a function. The second expression fails because our operational semantics disallows comparisons that involve function values. The goal of static type checking is to identify such expressions before we actually evaluate them. This way we can statically detect certain programming errors and provide a correctness guarantee for the evaluation, namely, that the evaluation will never get stuck in a dynamic type error. We refer to this feature of a programming language as a static type system.

Unfortunately, the problem of checking whether the evaluation of a given expression in our language will result in a dynamic type error is undecidable. In fact, this is true for any non-trivial programming language. There are two ways to deal with this dilemma: (1) we can require that the programmer provides some help to the type checking algorithm by annotating the program with type information and (2) we can allow the type checker to reject certain expressions as unsafe even though the expression could be safely evaluated. The type systems of most statically typed programming languages use a combination of (1) and
The crux in designing a static type system is to find a good balance between the annotation burden for the programmer, the restrictions imposed on what programs are considered safe, and the computational complexity of the actual type checking algorithm. We will explore some of these design choices.

A Simple Typed Language

We extend our language from class 11 with types. We choose syntax for types that is compatible with the TypedScript language, which is a typed extension of JavaScript. The abstract syntax of our new language is as follows:

\[
\begin{align*}
 n & \in \text{Num} \quad \text{numbers (double)} \\
 x & \in \text{Var} \quad \text{variables} \\
 b & \in \text{Bool} ::= \text{true} \mid \text{false} \quad \text{Booleans} \\
 \tau & \in \text{Typ} ::= \text{bool} \mid \text{number} \mid (x : \tau_1) \Rightarrow \tau_2 \quad \text{types} \\
 v & \in \text{Val} ::= n \mid b \mid \text{function} p (x : \tau) t e \quad \text{values} \\
 e & \in \text{Expr} ::= x \mid v \mid e_1 \text{ bop } e_2 \mid e_1 \ ? \ e_2 \ : \ e_3 \mid \text{expressions} \\
 \text{const} & ::= x = e_1 ; e_2 \mid e_1 (e_2) \\
 \text{bop} & \in \text{Bop} ::= + \mid * \mid === \mid !== \mid \&\& \mid || \quad \text{binary operators} \\
 p & ::= x \mid \epsilon \quad \text{function names} \\
 t & ::= : \tau \mid \epsilon \quad \text{return type annotations}
\end{align*}
\]

A type \( \tau \) is either one of the base types \text{bool} or \text{number}, or a function type \((x : \tau_1) \Rightarrow \tau_2\), where \( \tau_1 \) and \( \tau_2 \) are again types. A function type describes the signature of a function that has a parameter \( x \) of type \( \tau_1 \) and returns a value of type \( \tau_2 \). The name \( x \) of the parameter is irrelevant for most our discussion. We include it in the function type for the sake of compatibility with the TypedScript language.

The expression language is mostly identical to the language that we considered in class 11. The only difference is that the parameter of a function must now be annotated with a type. Moreover, functions can be annotated with optional return types. In fact, we will see that the return type annotation is mandatory for recursive functions. However, we do not enforce this property at the level of the grammar rules. Instead, we will enforce it using the typing rules, which we discuss below.

Parameter and return types are the only type annotations that the programmer is required to provide. In all other cases, our type checking algorithm will be able to automatically infer the type of an expression from the usages of operators and typed variables in the expression. In particular, variables that are introduced using \text{const} declarations do not have to be typed explicitly.

Typing Relation

We formalize our type checking algorithm in terms of a typing relation. The typing relation is defined similarly to the big-step structural operational semantic-
tics with dynamic scoping that we discussed in class 11. The main difference is that now types take the role of values. In particular, the typing relation is defined in terms of a *typing environment* $\Gamma : \text{Var} \rightarrow \text{Typ}$ that maps the free variables in an expression to their types. The typing relation is then denoted by the judgment form

$$\Gamma \vdash e : \tau$$

which reads “under typing environment $\Gamma$, the expression $e$ has type $\tau$”. The inference rules defining the typing relation are given in Figure 1.

It is instructive to compare the rules with the corresponding rules for the big-step operational semantics with dynamic scoping. Note that for the typing relation, the order in which the types for subexpressions are inferred is irrelevant. In fact, it is easy to see that the typing rules are deterministic, i.e., for any $\Gamma$ and $e$, there is at most one type $\tau$ such that $\Gamma \vdash e : \tau$. Thus, we can think of the typing relation $\Gamma \vdash e : \tau$ as a partial function that maps a typing environment $\Gamma$ and an expression $e$ to a type $\tau$. Also note that the typing relation uses static binding of variables rather than dynamic binding. This is because variables are only analyzed in their static scopes (i.e., they don’t occur in values that are stored in an environment and are extracted later to be analyzed in a different scope).

If a type $\tau$ exists such that $\Gamma \vdash e : \tau$, we say that $e$ is *well-typed* under $\Gamma$. If $e$ is a closed expression (i.e., the typing environment does not matter), then we simply say that $e$ is well-typed. If the typing relation fails to infer a type for a given $\Gamma$ and $e$, we say that $e$ has a type error under $\Gamma$.

Note that the static typing rules are much less permissive than the rules for the operational semantics with dynamic typing. For example, the following two expressions are not well-typed due to the absence of implicit type coercions in the typing relation:

$$3 + \text{true}$$

$$0 \ ? 1 : 2$$

The omission of type coercion is a deliberate design choice. Implicit type coercion is often a source of subtle bugs in programs that can be difficult to debug. Statically typed languages therefore often require the programmer to make type coercion explicit by inserting type casts or explicit calls to coercion functions.

Some expressions are not well-typed in our type system although they can be safely evaluated even without implicit type coercion. For example, the following conditional expression is not well-typed because the types of the two branches do not agree:

$$0 == 1 \ ? 1 : \text{false}$$

This is again a deliberate design choice. This time, the motivation is to reduce the complexity of the type checking algorithm. If we wanted the above expression to be well-typed, we would either have to make the typing relation nondeterministic or we would have to introduce a *supertype* of `number` and
\[\Gamma \vdash b : \text{bool} \quad \Gamma \vdash n : \text{number} \quad \text{TypeBool} \]
\[\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \text{bool} \quad \text{TypeANDOR} \quad \text{bop} \in \{\&\&, |\|\} \]
\[\Gamma \vdash e_1 \text{bop} e_2 : \text{bool} \quad \text{TypeANDOR} \]
\[\Gamma \vdash e_1 : \text{number} \quad \Gamma \vdash e_2 : \text{number} \quad \text{TypeARITH} \quad \text{bop} \in \{+, *\} \]
\[\Gamma \vdash e_1 \text{bop} e_2 : \text{number} \quad \text{TypeARITH} \]
\[\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau \quad \tau \neq (x : \tau_1) \Rightarrow \tau_2 \quad \text{TypeEQUAL} \quad \text{bop} \in \{==, !==\} \]
\[\Gamma \vdash e_1 \text{bop} e_2 : \text{bool} \quad \text{TypeEQUAL} \]
\[\Gamma \vdash e_d : \tau_d \quad \text{TypeConst} \quad \Gamma' = \Gamma[x \mapsto \tau_d] \quad \Gamma' \vdash e_b : \tau_b \]
\[\Gamma \vdash \text{const } x = e_d; e_b : \tau_b \quad \text{TypeConst} \]
\[\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau \quad \text{TypeIF} \quad \Gamma \vdash e_1 \quad e_2 \quad e_3 : \tau \]
\[\Gamma \vdash e_1 : (x : \tau') \Rightarrow \tau \quad \Gamma \vdash e_2 : \tau' \quad \text{TypeCall} \quad \Gamma \vdash e_1 (e_2) : \tau \]
\[\Gamma' = \Gamma[x \mapsto \tau] \quad \Gamma' \vdash e : \tau' \quad \text{TypeFunction} \quad \Gamma' = \Gamma[x \mapsto \tau] \quad \Gamma' \vdash e : \tau' \quad \text{TypeFunction} \]
\[\Gamma' \vdash \text{function } (x : \tau)(e : (x : \tau) \Rightarrow \tau') \quad \text{TypeFunction} \quad \Gamma' = \Gamma[x \mapsto \tau] \quad \Gamma' \vdash e : \tau' \quad \text{TypeFunctionAnn} \quad \Gamma' \vdash \text{function } (x : \tau) : \tau' e : (x : \tau) \Rightarrow \tau' \quad \text{TypeFunctionAnn} \quad \Gamma' = \Gamma[x_1 \mapsto \tau_1][x_2 \mapsto \tau_2] \quad \Gamma' \vdash e : \tau' \quad \tau_1 = (x_2 : \tau_2) \Rightarrow \tau' \quad \text{TypeFunctionRec} \quad \Gamma \vdash \text{function } x_1 (x_2 : \tau_2) : \tau' e : \tau_1 \quad \text{TypeFunctionRec} \]

Figure 1: Type checking rules
bool that describes both Num and Bool values. We will consider such type systems later when we discuss object-oriented languages.

Perhaps the most severe restriction of our type system is that all types are monomorphic, i.e., they are either base types or function types constructed from base types. This means that our type system does not support any notion of type parametrization. For example, consider the following expression in our language of class 11:

```plaintext
const id = function (x) x;
id(false) ? id(2) : id(1)
```

This expression can be safely evaluated in our dynamically typed semantics. In particular, no type coercions are necessary during evaluation. Nevertheless, there is no type $\tau$ such that the following expression is well-typed in our new language:

```plaintext
const id = function (x: $\tau$) x;
id(false) ? id(2) : id(1)
```

If we choose $\tau = \text{bool}$, then the calls $\text{id}(2)$ and $\text{id}(1)$ will yield static type errors. If we choose $\tau = \text{number}$, then the call $\text{id}(\text{false})$ will not be well-typed. Finally, if we choose $\tau = (y:\tau_1) \Rightarrow \tau_2$ for any $\tau_1$ and $\tau_2$, then none of the calls will be well-typed. We will see later that we can define a more expressive static type system in which there exists a type $\tau$ such that the above expression is well-typed. In fact, in this more expressive type system the type $\tau$ can be automatically inferred and does not need to be provided by the programmer.