Compiler Development (CMPSC 401)

Type Checking

Janyl Jumadinova

March 14, 2019
What is a Type?

This is the subject of some debate.

The consensus:
- A set of values.
- A set of operations on those values.

Type errors arise when operations are performed on values that do not support that operation.
What is a Type?

- This is the subject of some debate.
- The consensus:
  - A set of values.
  - A set of operations on those values.
What is a Type?

- This is the subject of some debate.
- The consensus:
  - A set of values.
  - A set of operations on those values.
- Type errors arise when operations are performed on values that do not support that operation.
Types of Type-Checking

- **Static type checking:**
  - Analyze the program during compile-time to prove the absence of type errors.
  - Never let bad things happen at runtime.

- **Dynamic type checking:**
  - Check operations at runtime before performing them.
  - More precise than static type checking, but usually less efficient.

- **No type checking:**
  - Throw caution to the wind!
The rules governing permissible operations on types forms a type system.

Strong type systems are systems that never allow for a type error.
- Java, Python, JavaScript, LISP, Haskell, etc.

Weak type systems can allow type errors at runtime.
- C, C++
Design Space of Types

- **Weak**
  - Machine code
  - Javascript
- **Strong**
  - C
  - C++
  - SML
  - Java
  - Scheme
- **Dynamic**
Types vs. AST

Types are not AST nodes!

- AST nodes may have a type field, however.
Types vs. AST

Types are not AST nodes!

- AST nodes may have a type field, however.

- **AST**: abstract representation of source program (including source program type info).
Types vs. AST

Types are not AST nodes!

- AST nodes may have a type field, however.

- **AST**: abstract representation of source program (including source program type info).

- **Types**: abstract representation of type semantics for type checking, inference, etc.
  - Can include information not explicitly represented in the source code, or may describe types in ways more convenient for processing.
How do we determine the type of an expression?

Think of process as **logical inference**.
Inferring Expression Types

- How do we determine the type of an expression?
- Think of process as **logical inference**.

```
  int
 /   \
+   +
|   |
int IntConstant  int IntConstant
|   |
137 42
```
Inferring Expression Types

- How do we determine the type of an expression?
- Think of process as **logical inference**.
Inferring Expression Types

- How do we determine the type of an expression?
- Think of process as logical inference.

```
bool x Identifier == bool y Identifier == bool true BoolConstant
```
Example from BCD book

```
Program → Funs
Funs → Fun
Funs → Fun Funs

Fun → TypeId (TypeIds) = Exp

TypeId → int id
TypeId → bool id

TypeIds → TypeId
TypeIds → TypeId , TypeIds

Exp → num
Exp → id
Exp → Exp + Exp
Exp → Exp = Exp
Exp → if Exp then Exp else Exp
Exp → id (Exps)
Exp → let id = Exp in Exp
```
A program is a list of function declarations.
The functions are all mutually recursive.
No function may be declared more than once.
Functions and variables have separate name spaces (defining a name in one space doesn’t affect the same name in the other space).
Each function declares its result type and the types and names of its arguments.
There may not be repetitions in the list of parameters for a function.
The body of a function is an expression.
Comparison is defined both on booleans and integers, but addition only on integers.
A program must contain a function called main, which has one integer argument and which returns an integer.
Execution of the program is by calling the main function and the result of this function call is the output of the program.
Example: Static Type Checking

- Need symbol tables that bind variables and functions to their types.
- Use two symbol tables, one for variables and one for functions.
- A variable is bound to one of the two types \texttt{int} or \texttt{bool}.
- A function is bound to its type, which consists of the types of its arguments and the type of its result.
- Function types are written as a parenthesised list of the argument types,
  \[(\texttt{int, bool}) \rightarrow \texttt{int}\]
Example: Static Type Checking

- The function for type checking expressions is called `CheckExp`.
- The symbol table for variables is given by the parameter `vtable`.
- The symbol table for functions is given by the parameter `ftable`.
- The function `error` reports a type error.
Example: Type checking of expressions

\[
\text{Check}_{\text{Exp}}(\text{Exp}, \text{vtable}, \text{itable}) = \text{case Exp of}
\]

<table>
<thead>
<tr>
<th>num</th>
<th>int</th>
</tr>
</thead>
</table>
| \text{id} | \begin{align*}
& \text{let } t = \text{lookup}(\text{vtable}, \text{getname}(\text{id})) \\
& \quad \text{if } t = \text{unbound} \\
& \quad \quad \text{then error(); int} \\
& \quad \text{else } t
\end{align*} |

\begin{align*}
\text{Check}_{\text{Exp}}(\text{Exp}_1, \text{vtable}, \text{itable}) \\
\text{Check}_{\text{Exp}}(\text{Exp}_2, \text{vtable}, \text{itable})
\end{align*}

\begin{align*}
\text{Check}_{\text{Exp}}(\text{Exp}_1, \text{vtable}, \text{itable}) \\
\text{Check}_{\text{Exp}}(\text{Exp}_2, \text{vtable}, \text{itable})
\end{align*}

\begin{align*}
\text{Check}_{\text{Exp}}(\text{Exp}_3, \text{vtable}, \text{itable})
\end{align*}

\begin{align*}
\text{id} (\text{Exps}) \\
& = \text{lookup}(\text{itable}, \text{getname}(\text{id})) \\
& \quad \text{if } t = \text{unbound} \\
& \quad \quad \text{then error(); int} \\
& \quad \text{else} \\
& \quad \quad ((t_1, \ldots, t_n) \rightarrow t_0) = t \\
& \quad \quad [t'_1, \ldots, t'_n] = \text{Check}_{\text{Exp}}(\text{Exps}, \text{vtable}, \text{itable}) \\
& \quad \quad \text{if } m = n \text{ and } t_1 = t'_1, \ldots, t_n = t'_n \\
& \quad \quad \text{then } t_0 \\
& \quad \quad \text{else error(); } t_0
\end{align*}

\begin{align*}
& \text{let } \text{id} = \text{Exp}_1 \\
& \text{in } \text{Exp}_2
\end{align*}

\begin{align*}
& \text{Check}_{\text{Exp}}(\text{Exp}_1, \text{vtable}, \text{itable}) \\
& \text{vtable}' = \text{bind}(\text{vtable}, \text{getname}(\text{id}), t_1) \\
& \text{Check}_{\text{Exp}}(\text{Exp}_2, \text{vtable}', \text{itable})
\end{align*}
Example: Type checking of expressions

\[
\text{Check}_{\text{Exps}}(\text{Exps}, vtable, ftable) = \text{case Exps of}
\]
\[
\begin{array}{|c|c|}
\hline
\text{Exp} & \text{[Check}_{\text{Exp}}(\text{Exp}, vtable, ftable)] \\
\hline
\text{Exp , Exps} & \text{Check}_{\text{Exp}}(\text{Exp}, vtable, ftable) \\
& \text{:: Check}_{\text{Exps}}(\text{Exps}, vtable, ftable) \\
\hline
\end{array}
\]
Example: Type checking of function declarations

\[
\text{Check}_{\text{Fun}}(\text{Fun, ftable}) = \text{case Fun of}
\]
\[
\begin{array}{ll}
\text{TypeId (TypeIds)} = \text{Exp} & (f, t_0) = \text{Get}_{\text{TypeId}}(\text{TypeId}) \\
& \text{vtable} = \text{Check}_{\text{TypeId}}(\text{TypeIds}) \\
& t_1 = \text{Check}_{\text{Exp}}(\text{Exp, vtable, ftable}) \\
& \text{if } t_0 \neq t_1 \\
& \text{then error()}
\end{array}
\]

\[
\text{Get}_{\text{TypeId}}(\text{TypeId}) = \text{case TypeId of}
\]
\[
\begin{array}{ll}
\text{int id} & (\text{getname(id)}, \text{int}) \\
\text{bool id} & (\text{getname(id)}, \text{bool})
\end{array}
\]

\[
\text{Check}_{\text{TypeId}}(\text{TypeIds}) = \text{case TypeIds of}
\]
\[
\begin{array}{ll}
\text{TypeId} & (x, t) = \text{Get}_{\text{TypeId}}(\text{TypeId}) \\
& \text{bind}(<\text{emptytable}, x, t>) \\
\text{TypeId, TypeIds} & (x, t) = \text{Get}_{\text{TypeId}}(\text{TypeId}) \\
& \text{vtable} = \text{Check}_{\text{TypeId}}(\text{TypeIds}) \\
& \text{if lookup(vtable, x) = unbound} \\
& \text{then bind(vtable, x, t)} \\
& \text{else error(); vtable}
\end{array}
\]
Limitations of Static Type Systems

- Static type systems are often incomplete.
  - There are valid programs that are rejected.
- Tension between the static and dynamic types of objects.
  - Static type is the type declared in the program source.
  - Dynamic type is the actual type of the object at runtime.
Soundness and Completeness

- Static type systems sometimes reject valid programs because they cannot prove the absence of a type error.
- A type system like this is called **incomplete**.
- Instead, try to prove for every expression that $\text{DynamicType}(E) \leq \text{StaticType}(E)$
- A type system like this is called **sound**.
Unfortunately, for most programming languages, it is provably impossible to have a sound and complete static type checker.

**Intuition:** Could build a program that makes a type error iff a certain Turing machine accepts a given string.

Type-checking equivalent to solving the halting problem!
It is difficult to build a good static type checker.
- Easy to have unsound rules.
- Impossible to accept all valid programs.

Goal: make the language as complete as possible with sound type-checking rules.
```java
class Base {
    Base clone() {
        return new Base;
    }
}

class Derived extends Base {
    Derived clone() {
        return new Derived;
    }
}

Is this safe?
```
Intuition

```java
class Base {
    Base clone() {
        return new Base;
    }
}

class Derived extends Base {
    Derived clone() {
        return new Derived;
    }
}
```

Base b = new Base;
Derived d = new Derived;
Base b2 = b.clone();
Base b3 = d.clone();
Derived d2 = b.clone();
Derived d3 = d.clone();

Base reallyD = new Derived;
Base b4 = reallyD.clone();
Derived d4 = reallyD.clone();

Is this safe?
Is this safe?

$f$ is an identifier.
\[ S \leftarrow e_0 : M \]

$f$ is a member function in class $M$.
\[ f \text{ has type } (T_1, \ldots, T_n) \to U \]
\[ S \leftarrow e_i : R_i \text{ for } 1 \leq i \leq n \]
\[ R_i \leq T_i \text{ for } 1 \leq i \leq n \]
\[ S \leftarrow e_0.f(e_1, \ldots, e_n) : U \]

This refers to the static type of the function.

$f$ has dynamic type
\[ (T_1, T_2, \ldots, T_n) \to V \]
and we know that
\[ V \leq U \]

so the rule is sound!
So what?

- Need to be very careful when introducing language features into a statically-typed language.
- Easy to design language features; hard to design language features that are type-safe.
- Type proof system can sometimes help detect these errors in the abstract.