Type Systems

Pierce Ch. 3, 8, 11, 15
Goals

Define the simple **language of expressions**
   A small subset of Lisp, with minor modifications

Define the **type system** of this language
   Mathematical definition using standard machinery: inference rules from formal logic

Understand the relationship between semantics and type systems
   And more generally, the role of types in programming languages
A Simple Subset of Lisp (with minor tweaks)

\[<E> ::= \text{const} \mid T \mid F \mid (\text{IF} \ <E> \ \text{THEN} \ <E> \ \text{ELSE} \ <E>) \mid (\text{INT} \ <E>) \mid (\text{EQ} \ <E> \ <E>) \mid (\text{PLUS} \ <E> \ <E>) \mid (\text{LESS} \ <E> \ <E>)\]

\text{const} represents a numeric atom for a natural number (0, 1, 2, ...)

T and F are literal atoms for “true” and “false” (not like real Lisp)

\text{IF-THEN-ELSE} is part of real Lisp. In our fake language, if the first expression evaluates to T, we evaluate the second expression; if the first expression evaluates to F, we evaluate the third expression; otherwise undefined

\text{INT, EQ, PLUS, LESS} were defined in Project 3
Other Ways to Define the Syntax

Inductive definition: the smallest set $S$ such that

\[
\{ \text{const, } T, \ F \} \subseteq S \\
\text{if } E \in S, \text{ then } (\text{INT } E) \in S \\
\text{if } E_1, E_2 \in S, \text{ then } \{ (\text{EQ } E_1 E_2), (\text{PLUS } E_1 E_2), (\text{LESS } E_1 E_2) \} \subseteq S \\
\text{if } E_1, E_2, E_3 \in S, \text{ then } (\text{IF } E_1 \text{ THEN } E_2 \text{ ELSE } E_3) \in S
\]

Now the same thing, written as inference rules (formal logic)

\[
\begin{align*}
\text{const} & \in S \\
T & \in S \\
F & \in S \\
E & \in S \\
E_1 & \in S \\
E_2 & \in S \\
E_1 & \in S \\
E_2 & \in S \\
E_3 & \in S
\end{align*}
\]

\[
\begin{align*}
(\text{INT } E) & \in S \\
(\text{EQ } E_1 E_2) & \in S \\
(\text{IF } E_1 \text{ THEN } E_2 \text{ ELSE } E_3) & \in S
\end{align*}
\]

If we have established the premises (above the line), we can derive the conclusion (below the line)
Language Semantics

Defined in Project 3 through function `eval`

“Running a program”: expression evaluates to value, which is an atom `T` or `F` or `const`

- `(IF F THEN 0 ELSE 31)` evaluates to `31`
- `(INT (PLUS 5 6))` evaluates to `T`

Run-time error: when `eval(E)` is `undefined`

- `(PLUS (EQ 1 2) 5)`
- `(IF 8 THEN 1 ELSE 2)`

Can we prevent some of these errors?
Type System and Type Checking

A type system can help us prove that certain programs are “good” without running them.

If a program is well-typed, it will not “go wrong” at run time.

Guarantees are only for \textit{some} run-time errors, \textbf{not all}:

E.g. \textit{cannot} assure the absence of “division by zero” or “array index out of bounds” or “\texttt{CAR} applied to an empty list” - they depend on \textit{particular values} from a type.

But \textit{can} catch “\texttt{PLUS} with a boolean operand” error.
Typed Expressions

Goal: without evaluating an expression, can we guarantee that its evaluation will not produce a run-time error? (static a.k.a. compile-time analysis)

Solution: define types, and establish a relationship between expressions and types

For our simple language

Type $\text{Bool} = \text{set of all expressions that evaluate to T or F}$
Type $\text{Nat} = \text{set of all expressions that evaluate to const}$

To determine that an expression $E$ has type $T$ (i.e., $E \in T$), will only look at the structure of $E$ but will not evaluate $E$
Typing Relation

Relation: \( \subseteq S \times \{ \text{Bool, Nat} \} \)

\( E : T \) is just another notation for \( E \in T \)

\[ \begin{align*}
T & : \text{Bool} \\
F & : \text{Bool} \\
\text{const} & : \text{Nat}
\end{align*} \]

\[ \begin{align*}
E_1 : \text{Bool} & \quad E_2 : T & \quad E_3 : T \\
\text{(IF } E_1 \text{ THEN } E_2 \text{ ELSE } E_3 \text{)} : T \\
\end{align*} \]

\[ \begin{align*}
E_1 : \text{T}_1 & \quad E_2 : \text{T}_2 \\
\text{EQ } E_1 E_2 \text{ : Bool} \\
\end{align*} \]

\[ \begin{align*}
E : T & \\
\text{(INT } E \text{)} : \text{Bool} \\
\end{align*} \]

\[ \begin{align*}
E_1 : \text{Nat} & \quad E_2 : \text{Nat} \\
\text{PLUS } E_1 E_2 \text{ : Nat} \\
\end{align*} \]

\[ \begin{align*}
E_1 : \text{Nat} & \quad E_2 : \text{Nat} \\
\text{LESS } E_1 E_2 \text{ : Bool} \\
\end{align*} \]
Example: Typing Derivation

(IF (INT 5) THEN 6 ELSE (PLUS 7 8)) : ?

```
5 : Nat
_______
(INT 5) : Bool 6 : Nat

7 : Nat 8 : Nat
_______
(PLUS 7 8) : Nat
```

(IF (INT 5) THEN 6 ELSE (PLUS 7 8)) : Nat

This structure is a derivation tree: the leaves are instances of axioms, the inner nodes are instances of non-axiom rules
More on the Typing Relation

E is **typable** (or **well-typed**) if exists T such that \( E : T \)

In our type system, each well-typed expression has one type; in general, an expression may have multiple types (e.g., when the type system has **subtypes**)

**Safety** (a.k.a. **soundness**) of a type system: a well-typed program will not have a run-time error

For our type system: for a well-typed \( E : T \) we know that \( \text{eval}(E) \) is defined and is a value of type \( T \)

This property does not work in the other direction: an expression which is not well-typed may or may have a run-time error (**conservative static analysis**)

\((\text{IF (INT 33) THEN 44 ELSE (PLUS T F)})\) is not well-typed but runs fine
Lists

\[ \langle E \rangle ::= \ldots \mid \text{NIL} \mid (\text{NULL} \ \langle E \rangle) \mid (\text{CONS} \ \langle E \rangle \ \langle E \rangle) \mid (\text{CAR} \ \langle E \rangle) \mid (\text{CDR} \ \langle E \rangle) \]

Semantics: a value now can also be a list value
Either \text{NIL}, or \text{CONS} applied to a value and a list value

Typing: need to add list types of the form “List (T)”
Example: List (List (Nat))
Note that lists will be homogeneous – all elements will have the same type. This is not the case for real Lisp – lists there are heterogeneous.
Typing Relation

NIL : List (T) for any T

\[
\begin{array}{c}
E_1 : T \\
E_2 : \text{List} (T)
\end{array}
\]
\[
\frac{\text{(CONS } E_1 E_2 \text{)} : \text{List} (T)}{E : \text{List} (T)}
\]

E : List (T)

\[
\frac{(\text{CAR } E) : T}{(\text{CONS } E_1 E_2) : \text{List} (T)}
\]

\[
\frac{(\text{CDR } E) : \text{List(T)}}{E : \text{List} (T)}
\]

\[
\frac{(\text{NULL } E) : \text{Bool}}{E : \text{List} (T)}
\]

Example 1: (CONS (NULL NIL) (CONS F NIL))
Example 2: (CONS F T)
Example 3: (NULL NIL)
Example 4: (CONS (NULL NIL) (CONS 8 NIL))
Brief Overview of Terminology

Polymorphism

Statically vs dynamically typed languages

Type safety vs language safety
Polymorphism

Poly = many, morph = form

A piece of code has multiple types

Example 1: **subtype polymorphism**
- An expression has multiple types
- Typical for object-oriented languages (class Y extends X)

Example 2: **parametric polymorphism**
- E.g. \( f(x) = x \) has types \( \text{Bool} \rightarrow \text{Bool}, \text{Nat} \rightarrow \text{Nat}, \ldots \)
- Use a **type parameter** \( T \); define type type \( T \rightarrow T \)
- Generics in C++ and Java – e.g. `Map<K,V>`
- ML and similar functional languages

Example 3: **coercion and overloading**
Coercion and Overloading

Automatic coercion (conversion) to another type is performed silently: e.g. in Java byte can be “widened” to short, int, long, float, or double

– E.g. in assignment conversion, the right-hand-side is converted to the type of the left-hand-side var

– E.g. numeric promotion converts operands of a numeric operator to a common type, e.g. for +

Overloading: multiple definitions of the same name e.g. in Java name + has several types:

- double × double → double
- long × long → long
- double → double
- long → long
- float × float → float
- int × int → int
- float → float
- int → int
Terminology

Statically typed language: expressions have static (compile-time) types, and we do static type checking

- **Goal**: prove the absence of certain type-related bad behaviors before running the program

Declared types: C/C++/Java/... (programmer gives types)

Inferred types: ML/Haskell – no programmer-declared types; compiler infers, based on use in operators/functions

Type safety: all bad behaviors of certain type-related kinds are excluded - e.g., Java, but not C (due to arbitrary typecasting in C)

Dynamically typed language: run-time checks to catch type-related bad behaviors (e.g. Lisp, Perl)

- E.g., in our projects, PLUS must be applied to numbers
Terminology

Want more than static type safety – want language safety
Cannot “break” the abstractions of the language (type-related and otherwise); e.g. no buffer overflows, seg faults, return address overriding, garbage values, etc.

Example: C is unsafe for many reasons, one of which is the lack of type safety: e.g., `double pi = 3.14; int* ptr = (int*) &pi; int x = *ptr;`
Other reasons: null pointers lead to seg faults (OS concept, not PL concept); buffer overflows lead to stack smashing & garbage values

Example: Java is safe – combination of static type safety & run-time checks. Static type safety ensures that an well-typed program will not do type-related “bad” things. Run-time checks catch things that cannot be caught statically via types: null pointers, array index out of bounds, etc.

Example: Lisp is safe – checks for type-related correctness (“operands of PLUS must be numbers”) and special “bad” values (e.g. divide by 0)