Type Systems

- Pierce Ch. 3, 8, 11, 15

A Simple Language

\[
\begin{align*}
\langle t \rangle ::= & \quad \text{true} \mid \text{false} \mid \text{if} \langle t \rangle \text{ then } \langle t \rangle \text{ else } \langle t \rangle \\
               & \mid 0 \mid \text{succ} \langle t \rangle \mid \text{pred} \langle t \rangle \mid \text{iszero} \langle t \rangle
\end{align*}
\]

- Simple untyped expressions
- Natural numbers encoded as: \text{succ} ... \text{succ} 0
  - E.g. \text{succ succ succ 0} represents 3
- \text{term}: a string from this language
  - To improve readability, we will sometimes write parentheses: e.g. \text{iszero (pred (succ 0))}

Semantics (informally)

- A term evaluates to a value
- Values are terms themselves
- Boolean constants: true and false
- Natural numbers: 0, succ 0, succ (succ 0), ...

Given a program (i.e., a term), the result of “running” this program is a boolean value or a natural number
- if false then 0 else succ 0 \rightarrow succ 0
- iszero (pred (succ 0)) \rightarrow true
- Problematic: succ true or if 0 then 0 else 0

Equivalent Ways to Define the Syntax

- Inductive definition: the smallest set S s.t.
  - \{ true, false, 0 \} \subseteq S
  - if \( t_1 \in S \), then \{ succ \( t_1 \), pred \( t_1 \), iszero \( t_1 \) \} \subseteq S
  - if \( t_1, t_2, t_3 \in S \), then if \( t_1 \) then \( t_2 \) else \( t_3 \) \in S
- Same thing, written as inference rules

\[
\begin{array}{ccc}
\text{true} & \text{false} & \text{0} \\
\text{succ} \text{ } \text{ } \text{t}_1 & \text{pred} \text{ } \text{ } \text{t}_1 & \text{iszero} \text{ } \text{ } \text{t}_1 \\
\text{if} \text{ } \text{ } \text{t}_1 & \text{then} \text{ } \text{ } \text{t}_2 & \text{else} \text{ } \text{ } \text{t}_3 \\
\text{t}_1 & \text{t}_2 & \text{t}_3 \\
\text{if} \text{ } \text{ } \text{t}_1 & \text{then} \text{ } \text{ } \text{t}_2 & \text{else} \text{ } \text{ } \text{t}_3 \\
\text{t}_1 & \text{t}_2 & \text{t}_3 \\
\end{array}
\]

Inductive Proofs

- Structural induction – used very often
- Suppose \( P \) is a predicate over terms (i.e., a function mapping elements of \( S \) to truth values)
  - When \( P(t) \) is true, we will just write \( P(t) \)
  - For each term \( t \), let \( t \) be its immediate subterms. Suppose we can prove that
    - Whenever \( P(t_1) \) for all \( t_1 \), we also have \( P(t) \)
    - For terms without subterms, \( P(t) \) holds
  - This means that \( P(t) \) for all terms in \( S \)
Semantics: Why?

- We need to define the semantics before we can discuss type systems
- The semantics defines the difference between "good" and "bad" programs
- A type system can help us prove that certain programs are "good", for all possible inputs
- Safety (a.k.a. soundness) of a type system: if a program is well-typed, it will not "go wrong"
  - But only for certain bad behaviors: e.g. a type system typically cannot assure the absence of "division by zero" or "array index out of bounds"

Semantics: How?

- Initial state: the term whose meaning we are trying to determine
  - i.e., the expression we are trying to evaluate
- One of two things can happen:
  - We reach a state (i.e. a term) which is a semantic value
  - We get stuck
- All of this depends on what we consider to be the set of semantic values

Evaluation Relation: Booleans

- Relation $\rightarrow \subseteq S \times S$ defined with inference rules
  - Just a way of writing an inductive definition

| if true then $t_2$ else $t_3$ $\rightarrow t_2$ |
| if false then $t_2$ else $t_3$ $\rightarrow t_3$ |
| $t_1 \rightarrow t'_1$ |
| if $t_1$ then $t_2$ else $t_3$ $\rightarrow$ if $t'_1$ then $t_2$ else $t_3$ |

- These rules get instantiated with concrete terms - to get rule instances

Semantics (formally)

- The domain of values (a subset of the terms)
  - $<v> ::= <bv> | <nv>$ values
  - $<bv> ::= true | false$ boolean values
  - $<nv> ::= 0 | succ <nv>$ numeric values
- Operational semantics defined by an evaluation relation on terms: $t \rightarrow t'$
  - $\rightarrow$ is a binary relation: $\rightarrow \subseteq S \times S$
  - $t \rightarrow t'$ means "$t$ evaluates to $t'$ in one step"
  - Thus, "small-step" operational semantics

Example

if true then 
  (if (if false then false else false) then true 
  else false) 
else 
  true $\rightarrow ?$ (value i.e. term that is true or false)

Step 1: ... $\rightarrow$ if (if false then false else false) then true else false
Step 2: if false then false else false $\rightarrow$ false
Step 3: if (if false then false else false) then true else false $\rightarrow$ if false then true else false
Step 4: if false then false else false $\rightarrow$ false
More on the Evaluation Relation

- We can generalize to the natural numbers by adding more inference rules
  - Will not go into these details here
- A key issue: what if we reach a term that cannot be evaluated anymore (no inference rule applies), but the term is not a semantic value?
  - Examples: if 0 then 0 else 0 and pred false
  - There is no inference rule that can be used to make "the next step"
- We get "stuck" – i.e. have a run-time error: the program has reached a meaningless state

Typed Expressions

- Goal: without evaluating a term, can we guarantee that it will not get stuck?
  - Idea: define types, and establish a relationship between terms and types
  - For our simple example:
    - Type Bool, which is the set of all terms that evaluate to a boolean value
    - Type Nat, which is the set of all terms that evaluate to a numeric value
  - To determine that a term t has type T (i.e., t \in T), we will only look at the structure of t (i.e., will do a compile-time analysis)

Typing Relation

- Relation \subseteq S \times \{ Bool, Nat \}
  - \( t : T \) is the same as \( t \in T \)
- \begin{align*}
  true & : Bool \\
  false & : Bool \\
  0 & : Nat \\
  \text{if } t_1 \text{ then } t_2 \text{ else } t_3 & : T \\
  \text{succ } t_1 & : Nat \\
  \text{pred } t_1 & : Nat \\
  \text{iszero } t_1 & : Bool
\end{align*}

Example: Typing Derivation

- if (iszero 0) then 0 else (succ 0) : ?
- \begin{align*}
  & 0 : Nat \\
  & \text{iszero } 0 : \text{Bool} \\
  & 0 : Nat \\
  & \text{succ } 0 : \text{Nat}
\end{align*}
  - This structure is a derivation tree: the leaves are instances of axioms, the inner nodes are instances of inference rules with premises

More on the Typing Relation

- A term t is typable (or well typed) if there is some T such that \( t : T \)
  - In this particular simple type system, each term has at most one type
  - In general, a term may have multiple types (e.g. when the type system has subtypes)
- Progress: A well-typed term will not be stuck: it either is a value, or it can take a step according to the evaluation rules
- Preservation: If a well-typed term takes a step of evaluation, the result is also well typed

More on the Typing Relation

- Safety = Progress + Preservation
  - Safety (a.k.a. soundness) of a type system: if a program is well-typed, it will not "go wrong"
  - For this type system: a well-typed term \( t : T \) will not get stuck
  - And will evaluate to a value of type T
  - This property does not work in the other direction: a term which is not well typed may or may not get stuck (conservative analysis)
  - if (iszero 0) then 0 else false
  - if true then 0 else false
An Extended Simple Language

\[ \langle t \rangle ::= \text{true} \mid \text{false} \mid \text{if} \langle t \rangle \text{ then } \langle t \rangle \text{ else } \langle t \rangle \mid 0 \mid \text{succ} \langle t \rangle \mid \text{pred} \langle t \rangle \mid \text{iszero} \langle t \rangle \mid \langle\langle t, t\rangle\rangle \mid \langle t \rangle.1 \mid \langle t \rangle.2 \]

- Pairs: \text{pairing} (, ) and \text{projection} 1/.2
- Need to add \textit{pair values} to the semantics
  - \[ \langle v \rangle ::= \langle b v \rangle \mid \langle n v \rangle \mid \{ \langle v \rangle, \langle v \rangle \} \]
- Generalization to n-tuples is trivial
- For typing: need to add pair types \( T_1 \times T_2 \)
  - E.g. \( \text{Bool} \times \text{Nat}, \text{Nat} \times \text{Nat}, \text{etc.} \)

Typing Relation Again

- No surprises here ...
  - \[
  \begin{array}{c}
  t_1 : T_1 \\
  t_2 : T_2 \\
  \{ t_1, t_2 \} : T_1 \times T_2 \\
  \end{array}
  \]
  - \[
  \begin{array}{c}
  t_1 : T_1 \times T_2 \\
  t_1.1 : T_1 \\
  t_1.2 : T_2 \\
  \end{array}
  \]

- \{ \text{if (iszero 0) then 0 else (succ 0)}, \text{true} \}.2 : ?
  - \{ \ldots \} : \text{Nat} \times \text{Bool}
  - \{ \ldots \}.2 : \text{Bool}

Records

\[ \langle t \rangle ::= \ldots \mid \{ l_1 \text{=} \langle t \rangle_1, l_2 \text{=} \langle t \rangle_2, \ldots, l_n \text{=} \langle t \rangle_n \} \mid \langle t \rangle.1 \]

- Example: \{ \text{sum} \text{=} \text{succ 0}, \text{overdraft} \text{=} \text{true} \}
- Labels \( l_i \) are from some pre-defined set of labels
  - In any term, all labels must be different
- In the semantics, introduce \textit{record values}
- In the type system, introduce \textit{record types}
  \( \{ l_1: T_1, l_2: T_2, \ldots, l_n: T_n \} \)
  - E.g. \{ \text{sum: Nat}, \text{overdraft: Bool} \}

Typing Relation

- Similar to the handling of tuples
  - \[
  \begin{array}{c}
  t_1 : T_1 \\
  t_2 : T_2 \\
  \ldots \\
  t_n : T_n \\
  \{ l_1 \text{=} t_1, l_2 \text{=} t_2, \ldots, l_n \text{=} t_n \} : \{ l_1: T_1, l_2: T_2, \ldots, l_n: T_n \}
  \end{array}
  \]
  - \[
  \begin{array}{c}
  t_1 : \{ l_1: T_1, l_2: T_2, \ldots, l_n: T_n \} \\
  t_1.l_k : T_k \\
  \end{array}
  \]

- \{ \text{sum} \text{=} \text{succ 0}, \text{overdraft} \text{=} \text{true} \}.\text{sum} : ?
  - \{ \ldots \} : \{ \text{sum: Nat}, \text{overdraft: Bool} \}
  - \{ \ldots \}.\text{sum} : \text{Nat}

Ordering of Labels

- Consider \{ \text{sum} \text{=} \text{succ 0}, \text{overdraft} \text{=} \text{true} \} and \{ \text{overdraft} \text{=} \text{true}, \text{sum} \text{=} \text{succ 0} \}
  - Are they the same value?
- Consider \{ \text{sum: Nat}, \text{overdraft: Bool} \} and \{ \text{overdraft: Bool}, \text{sum: Nat} \}
  - Are they the same type?
- In our type system, labels are ordered
  - Similarly to tuples: \( (0, \text{true}) \) is not \( (\text{true}, 0) \)
  - Will this typecheck in C?
  - \[
  \begin{array}{c}
  \text{struct} (\text{x: int}, \text{y: int}); \text{a:b: c:a:}
  \\
  \text{a.x = 1; a.y = 2; b = a; c = a:}
  \end{array}
  \]

Lists

\[ \langle t \rangle ::= \ldots \mid \text{nil}\langle T \rangle \mid \text{cons}\langle T \rangle \langle t \rangle \langle t \rangle \mid \text{isnil}\langle T \rangle \langle t \rangle \mid \text{head}\langle T \rangle \langle t \rangle \mid \text{tail}\langle T \rangle \langle t \rangle \]

- Example: \text{cons}[\text{Bool}] (\text{isnil}[\text{Nat}: \text{Bool}]) \{ \text{nil}[\text{Nat}: \text{Bool}] \text{false nil}[\text{Bool}] \}
  - The value is a list of size 2: \text{cons}[\text{Bool}] \text{true (cons[Bool] false nil[Bool]) i.e. (true false)}
  - In the semantics: \textit{list values}
    - \[ \langle v \rangle ::= \ldots \mid \text{nil}\langle T \rangle \mid \text{cons}\langle T \rangle \langle v \rangle \langle v \rangle \]
    - In the type system: \textit{list types}
      - \text{List} \( T \) - e.g. \text{List} (\text{List} \text{Nat}: \text{Nat})
Typing Relation

\[
\begin{align*}
\text{nil}[T_i] & : \text{List } T_1 \\
t_1 : T_1 & \quad t_2 : \text{List } T_1 \\
\text{cons}[T_j] t_1 t_2 & : \text{List } T_1 \\
t_1 : \text{List } T_1 & \\
isnil[T_i] t_1 & : \text{Bool}
\end{align*}
\]

- Example 1: \(\text{cons}[\text{Bool}] \) (isnil[\text{Nat} \times \text{Bool}]) (\text{cons}[\text{Bool}] \) false \(\text{nil}[\text{Nat} \times \text{Bool}]\)
- Example 2: \(\text{cons}[\text{Bool}] \) false true
- Example 3: isnil[\text{Bool}] \(\text{nil}[\text{Nat} \times \text{Bool}]\)

Subtypes

- Subtypes play an important role in many languages (e.g. object-oriented ones)
- \(S\) is a subtype of \(T\), written \(S <: T\), if any term of type \(S\) can be safely used in any situation where a term of type \(T\) is expected
- Principle of safe substitution
  \[
  \Gamma \vdash t : S \\
  \Gamma \vdash t : T
  \]
- Simple interpretation is that the elements of \(S\) form a subset of the elements of \(T\)
- We will define the subtype relation \(<:\) with the help of inference rules

Let Bindings

\[
\langle t \rangle ::= ... \mid \text{id} = \langle t \rangle \in \langle t \rangle
\]
- Give names to sub-expressions
  - let \(z=\text{true}\) in \(\text{cons}[\text{Bool}] z (\text{cons}[\text{Bool}] z \text{nil}[\text{Bool}])\)
- Semantics: evaluate the first expr, "bind" \(z\) to that value, and evaluate the second expr
- Use a type environment \(\Gamma\) (a.k.a. typing context)
  - Sequence of (name,type) pairs
  - \(\Gamma, x:T\) means "\(\Gamma\) appended with the pair \((x:T)\)"
  - Name \(x\) should not already be bound by \(\Gamma\)
- Ternary typing relation: \(\Gamma, t : T\)
  - "Term \(t\) has type \(T\) under the bindings in \(\Gamma\)"

Extended Typing Relation

- Need to include \(\Gamma\) in all rules; e.g.
  \[
  \begin{align*}
  \Gamma \vdash \text{true} & : \text{Bool} \\
  \Gamma, x: T & \vdash t_1 : T_1 \\
  \Gamma & \vdash \text{cons}[T_i] t_1 t_2 : \text{List } T_1
  \end{align*}
  \]
- \(\Gamma\) also needed for functions and function applications (function body should be evaluated under bindings for the function parameters)
  - But, we have no time for this discussion
- In this generalized type system, as before, each term has at most one type, and a well-typed term will not get stuck (safety)

Subtype Relation

\[
\begin{align*}
S <: S & \quad \text{reflexivity} \\
S <: U \quad U <: T & \quad \text{transitivity}
\end{align*}
\]

- \(S <: \text{Top}\) top type
- \(S <: S\) subsumption rule
- \(\begin{align*}
S_1 <: T_1 & \quad S_2 <: T_2 & \quad ... & \quad S_n <: T_n \\
\{ l_1:S_1, l_2:S_2, ..., l_n:S_n \} <: \{ l_1:T_1, l_2:T_2, ..., l_n:T_n \}
\end{align*}\)
- Depth subtyping for records
- \(\begin{align*}
\{ l_1:T_1, l_2:T_2, ..., l_n:T_n, l_{n+1}:T_{n+1} \} <: \{ l_1:T_1, l_2:T_2, ..., l_n:T_n \}
\end{align*}\)
- Width subtyping for records

Example: \((x:\text{Nat})\) is the set of all records that have a field \(x:\text{Nat}\), and some other fields. \((x:\text{Nat}, y:\text{Bool})\) is the set of all records that have a field \(x:\text{Nat}\), a field \(y:\text{Bool}\), and some other fields. Thus, \((x:\text{Nat}, y:\text{Bool}) <: (x:\text{Nat})\)
Should the Order of Labels Matter?

- The rule says that the order of labels (fields) in a record does not matter: e.g. \( \{ x : \text{Nat}, y : \text{Bool} \} \) is a subtype of \( \{ y : \text{Bool}, x : \text{Nat} \} \) and vice versa.
- Problem: this is bad for run-time performance.
  - If we fix the order at compile time, we would know, at compile time, the offset of the field with label \( l_n \) - allows efficient access for \( t.l_n \).
  - But with permutation, at run time need to "search" in memory for the actual location of \( l_n \).

Functions and Subtypes

- Function types: \( T_1 \rightarrow T_2 \)
  - For a term of type \( T_1 \), the result of applying the function on this term is of type \( T_2 \).
  - Subtyping: contravariant for the parameter, covariant for the result.

\[
\begin{align*}
    S_1 &<: T_1 \\
    S_2 &<: T_2 \\
    S_1 \rightarrow S_2 &<: T_1 \rightarrow T_2
\end{align*}
\]

- Function \( f \) of type \( S_1 \rightarrow S_2 \) accepts an argument of \( S_1 \), so it should be OK with an argument of \( T_1 \). Returns a value of \( S_2 \), so \( f(\ldots) \) can be used anywhere where \( T_2 \) is expected. So, \( f \) is also of type \( T_1 \rightarrow T_2 \).

Casting

- \( (T) t \) in Java and C++
  - Up-cast: a term is "forced" to a supertype of the type the typechecker would choose for it.
    \[
    \begin{align*}
    \Gamma \vdash t : T \\
    \Gamma \vdash (T) t : T
    \end{align*}
    \]
  - If \( \Gamma \vdash t : S \) and \( S <: T \), use this and the subsumption rule to derive \( \Gamma \vdash (T) t : T \).
  - Down-cast: force a type that cannot be determined statically.
    - The programmer says to the typechecker: "I know this will be the type; trust me!"
    - "trust but verify" e.g. run-time checks in Java.

Terminology

- Statically typed language: compile-time analyses
  - Prove the absence of certain type-related bad run-time behaviors (C, C++, Java, ML, Haskell,...)
    - Type safety: all bad behaviors of certain kinds are excluded (e.g. Java, but not C).
- Dynamically typed language: run-time checks to catch bad behaviors (e.g. Lisp, Scheme, Perl).
- Language safety: cannot "break" the fundamental abstractions (type-related and otherwise); e.g. no buffer overflows, seg faults, return address overriding, garbage values due to type errors, etc.
  - C: unsafe; Java: safe, static+dynamic checking; Lisp: safe, dynamic checking.