Type Systems

- Pierce Ch. 3, 8, 11, 15

A Simple Language

<t> ::= **true** | **false** | **if** <t> **then** <t> **else** <t>

| **0** | **succ** <t> | **pred** <t> | **iszero** <t>

- Simple untyped expressions
  - Natural numbers encoded as **succ** … **succ 0**
    - E.g. **succ succ succ 0** represents 3
- *term*: a string from this language
  - To improve readability, we will sometime write parentheses: e.g. **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** → **succ 0**
  - **iszero (pred (succ 0))** → **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** } ⊆ S
  - if $t_1 \in S$, then {**succ** t_1, **pred** t_1, **iszero** t_1 } ⊆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

$\boxed{\textbf{true} \in S}$ $\boxed{\textbf{false} \in S}$ $\boxed{\textbf{0} \in S}$ *axioms (no premises)*

$$\frac{t_1 \in S}{\textbf{succ } t_1 \in S} \qquad \frac{t_1 \in S}{\textbf{pred } t_1 \in S} \qquad \frac{t_1 \in S \quad t_2 \in S \quad t_3 \in S}{\textbf{if } t_1 \textbf{ then } t_2 \textbf{ else } t_3 \in S}$$

$$\frac{t_1 \in S}{\textbf{iszero } t_1 \in S}$$

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

Why Does This Matter?

- Key property: for any t∈S, one of three things must be true:
  - It is a constant (i.e., derived from an axiom)
  - It is of the form **succ** t_1, **pred** t_1, or **iszero** t_1 where t_1 is some smaller term
  - It is of the form **if** t_1 **then** t_2 **else** t_3 where t_1, t_2, and t_3 are some smaller terms
- The inference rules make this explicit, and make it easy for us to have
  - Inductive definitions of functions over S
  - Inductive proofs of properties of S

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_i be its immediate subterms. Suppose we can prove that
  - Whenever $P(t_i)$ for all t_i , 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?

- **Operational semantics** in the general sense: imagine an abstract machine
  - Some notion of the state of this machine
  - Transition function: given the current state, what is the next state?
    - It is possible that the machine gets "stuck" – there is no valid transition
- The semantics we will define for this simple language is a specific form of "small-step" operational semantics
  - state = term; transition = term simplification
  - Later will discuss "big-step" semantics

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

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 → t'**
  - → is a binary relation: → ⊆ S × S
  - t → t' means "t evaluates to t' in one step"
    - Thus, "small-step" operational semantics

Evaluation Relation: Booleans

- Relation → ⊆ S×S defined with inference rules
  - Just a way of writing an inductive definition

$$\textbf{if true then } t_2 \textbf{ else } t_3 \rightarrow t_2$$

$$\textbf{if false then } t_2 \textbf{ else } t_3 \rightarrow t_3$$

$$\frac{t_1 \rightarrow t'_1}{\textbf{if } t_1 \textbf{ then } t_2 \textbf{ else } t_3 \rightarrow \textbf{if } t'_1 \textbf{ then } t_2 \textbf{ else } t_3}$$

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

Example

if true then
  (if (if false then false else false) then
    true
  else
    false)
else
  true → ? (value i.e. term that is **true** or **false**)

Step 1: ... → **if (if false then false else false) then true else false**
Step 2: **if false then false else false → false**
Step 3: **if (if false then false else false) then true else false → if false then true else false**
Step 4: **if false then true else false → 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 : $\subseteq S \times \{ \text{Bool}, \text{Nat} \}$
  - **t : T** is the same as $t \in T$

$$\boxed{\textbf{true} : \text{Bool}} \qquad \boxed{\textbf{false} : \text{Bool}} \qquad \boxed{\textbf{0} : \text{Nat}}$$

$$\boxed{\frac{t_1 : \text{Bool} \qquad t_2 : T \qquad t_3 : T}{\textbf{if } t_1 \textbf{ then } t_2 \textbf{ else } t_3 : T}}$$

$$\boxed{\frac{t_1 : \text{Nat}}{\textbf{succ } t_1 : \text{Nat}}} \qquad \boxed{\frac{t_1 : \text{Nat}}{\textbf{pred } t_1 : \text{Nat}}} \qquad \boxed{\frac{t_1 : \text{Nat}}{\textbf{iszero } t_1 : \text{Bool}}}$$

Example: Typing Derivation

- **if (iszero 0) then 0 else (succ 0)** : ?

$$\frac{\dfrac{\dfrac{}{\textbf{0} : \text{Nat}}}{\textbf{iszero 0} : \text{Bool}} \qquad \dfrac{}{\textbf{0} : \text{Nat}} \qquad \dfrac{\dfrac{}{\textbf{0} : \text{Nat}}}{\textbf{succ 0} : \text{Nat}}}{\textbf{if (iszero 0) then 0 else (succ 0)} : \text{Nat}}$$

- 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

<t> ::= **true** | **false** | **if** <t> **then** <t> **else** <t>

| **0** | **succ** <t> | **pred** <t> | **iszero** <t>

| **{** <t> **,** <t> **}** | <t> **.1** | <t> **.2**

- Pairs: *pairing* { , } and *projection* .1/.2
  - Need to add *pair values* to the semantics
    - **<v> ::= <bv> | <nv> | { <v>, <v> }**
  - Generalization to n-tuples is trivial
- For typing: need to add *pair types* $T_1 \times T_2$
  - E.g. Bool × Nat, Nat × Nat, etc.

Typing Relation Again

- No surprises here …

$$\frac{t_1 : T_1 \quad t_2 : T_2}{\{ t_1 , t_2 \} : T_1 \times T_2}$$

$$\frac{t_1 : T_1 \times T_2}{t_1.1 : T_1} \qquad \frac{t_1 : T_1 \times T_2}{t_1.2 : T_2}$$

- **{if (iszero 0) then 0 else (succ 0),true}.2** : ?
  - **{ … }** : Nat × Bool
  - **{ … }.2** : Bool

Records

<t> ::= … | **{** l_1=<t>$_1$ **,** l_2=<t>$_2$ **,…,** l_n=<t>$_n$ **}** | <t>**.l**

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

Typing Relation

- Similar to the handling of tuples

$$\frac{t_1 : T_1 \quad t_2 : T_2 \quad \ldots \quad t_n : T_n}{\{ l_1{=}t_1, l_2{=}t_2, \ldots, l_n{=}t_n \} : \{ l_1{:}T_1, l_2{:}T_2, \ldots, l_n{:}T_n \}}$$

$$\frac{t_1 : \{ l_1{:}T_1, l_2{:}T_2, \ldots, l_n{:}T_n \}}{t_1.l_k : T_k}$$

- **{sum=succ 0,overdraft=true}.sum** : ?
  - **{ … }** : { sum:Nat , overdraft:Bool }
  - **{ … }.sum** : Nat

Ordering of Labels

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

Lists

<t> ::= … | **nil[**<T>**]** | **cons[**<T>**]** <t> <t>

| **isnil[**<T>**]** <t> | **head[**<T>**]** <t> | **tail[**<T>**]** <t>

- Example: **cons[Bool] (isnil[Nat×Bool] nil[Nat×Bool]) (cons[Bool] false nil[Bool])**
  - The value is a list of size 2: **cons[Bool] true (cons[Bool] false nil[Bool])** i.e. (true false)
- In the semantics: *list values*
  - **<v> ::= … | nil[**<T>**] | cons[**<T>**]** <v> <v>
- In the type system: *list types*
  - List T – e.g. List (List Nat×Nat)

Typing Relation

$\mathbf{nil}[T_1] : \text{List } T_1$

$$\frac{t_1 : T_1 \qquad t_2 : \text{List } T_1}{\mathbf{cons}[T_1]\ t_1\ t_2 : \text{List } T_1}$$

$$\frac{t_1 : \text{List } T_1}{\mathbf{isnil}[T_1]\ t_1 : \text{Bool}}$$

$$\frac{t_1 : \text{List } T_1}{\mathbf{head}[T_1]\ t_1 : T_1}$$

$$\frac{t_1 : \text{List } T_1}{\mathbf{tail}[T_1]\ t_1 : \text{List } T_1}$$

- Example 1: **cons[Bool] (isnil[Nat×Bool] nil[Nat×Bool]) (cons[Bool] false nil[Bool])**
- Example 2: **cons[Bool] false true**
- Example 3: **isnil[Bool] nil[Nat×Bool]**

Let Bindings

<t> ::= … | **let id** = <t> **in** <t>

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

Typing Relation

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma, x{:}T_1 \vdash t_2 : T_2}{\Gamma \vdash \mathbf{let}\ x = t_1\ \mathbf{in}\ t_2 : T_2}$$

$$\frac{x{:}T \in \Gamma}{\Gamma \vdash x : T}$$

- **let z=true in cons[Bool] z (cons[Bool] z nil[Bool])** : ?
- $\varnothing \vdash$ **true** : Bool
- z:Bool $\vdash$ **cons[Bool] z (cons[Bool] z nil[Bool])** : ?
- z:Bool $\vdash$ **z** : Bool z:Bool $\vdash$ **nil[Bool]** : List Bool
- z:Bool $\vdash$ **cons[Bool] z nil[Bool]** : List Bool
- z:Bool $\vdash$ **cons[Bool] z (cons[Bool] z nil[Bool])** : List Bool
- $\varnothing \vdash$ **let z=true in cons[Bool] z (cons[Bool] z nil[Bool])** : List Bool
  - Note: $\varnothing \vdash t : T$ is typically written simply as $\vdash t : T$

Extended Typing Relation

- Need to include Γ in all rules ; e.g.

$\Gamma \vdash$ **true** : Bool

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma \vdash t_2 : \text{List } T_1}{\Gamma \vdash \mathbf{cons}[T_1]\ t_1\ t_2 : \text{List } T_1}$$

- Γ 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)

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

$$\frac{\Gamma \vdash t : S \qquad S <: T}{\Gamma \vdash t : T}$$ *subsumption rule*

- 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

Subtype Relation

$S <: S$ *reflexivity*

$S <: \text{Top}$ *top type*

$$\frac{S <: U \qquad U <: T}{S <: T}$$ *transitivity*

$$\frac{S_1 <: T_1 \qquad S_2 <: T_2 \qquad \ldots \qquad S_n <: T_n}{\{ l_1{:}S_1, l_2{:}S_2, \ldots, l_n{:}S_n \} <: \{ l_1{:}T_1, l_2{:}T_2, \ldots, l_n{:}T_n \}}$$

depth subtyping for records

$$\{ l_1{:}T_1, l_2{:}T_2, \ldots, l_n{:}T_n, l_{n+1}{:}T_{n+1} \} <: \{ l_1{:}T_1, l_2{:}T_2, \ldots, l_n{:}T_n \}$$

width subtyping for records

Example: {x:Nat} is the set of all records that have a field x:Nat, and some other fields. {x:Nat,y:Bool} is the set of all records that have a field x:Nat, a field y:Bool, and some other fields. Thus, {x:Nat,y:Bool} <: {x:Nat}

Should the Order of Labels Matter?

$$\frac{\{ k_1{:}S_1, \ldots, k_n{:}S_n \} \text{ is a permutation of } \{ l_1{:}T_1, \ldots, l_n{:}T_n \}}{\{ k_1{:}S_1, \ldots, k_n{:}S_n \} <: \{ l_1{:}T_1, \ldots, l_n{:}T_n \}}$$

- The rule says that the order of labels (fields) in a record does not matter: e.g. {x:Nat,y:Bool} is a subtype of {y:Bool,x: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: $T_1 \to T_2$
  - 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

$$\frac{T_1 <: S_1 \qquad S_2 <: T_2}{S_1 \to S_2 \ <: \ T_1 \to T_2}$$

  - Function f of type $S_1 \to S_2$ 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(…) can be used anywhere where T_2 is expected. So, f is also of type $T_1 \to T_2$

Tuples and Lists

- n-tuples can be thought of as a special case of records with labels 1, 2, …, n
  - Essentially, same typing rules
- Lists

$$\frac{S_1 <: T_1}{\text{List } S_1 <: \text{List } T_1}$$

  - Allows the creation of heterogeneous lists: e.g. cons[{x:Nat}] {x=0} (cons[{x:Nat,y:Bool}] {x=0,y=true} nil[{x:Nat,y:Bool}])
  - For the inner expression: cons … : List {x:Nat,y:Bool}
  - Subsumption rule: give it type List {x:Nat}
  - Only then we can type the outer cons …

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

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash (T)\, t : T}$$

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

$$\frac{\Gamma \vdash t : S}{\Gamma \vdash (T)\, t : T}$$

  - 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

Polymorphism

- Poly = many, morph = form
- A piece of code has multiple types
- Example 1: subtype polymorphism
  - Subsumption rule: a term has multiple types
  - Typical for object-oriented languages
- Example 2: parametric polymorphism
  - E.g. f(x)=x has types Bool→Bool, Nat→Nat, …
  - Use a type parameter T and type T→T
    - Examples: generics in C++ and Java, ML-style polymorphism in functional languages
- Example 3: ad hoc polymorphism - e.g. overloading

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