\(\lambda\)-calculus and Types

- \(\lambda\) calculus is as expressive as Turing machines
- Can encode lists, numbers, Booleans, recursion, and other interesting data structures using it.
- However, thus far it imposes no structure on how terms are used
- To simplify program structure, it is useful to have a language with types.
Goals

- Understand types and type systems formally:
  - Many languages have informal descriptions of their type system
    - prone to error
  - Failure to have a formal type system can lead to false claims about a language’s safety
  - Formal systems allow formal proofs of safety
  - Can lead to a better informal understanding of the expressive power of a language

- Understand how types influence the expressive power of a language
  - Useful for language design and implementation
Types

- A program variable can assume a range of values during the execution of the program.
- An upper bound of such a range is called a type of the variable:
  - A variable of type “bool” is assumed to hold only Boolean values.
  - If x has type “bool” then “not(x)” has a sensible meaning in any context in which it is used.
Types and Languages

- Untyped Languages (e.g. $\lambda$-calculus)
  - The language does not restrict the range of values for a given variable.
  - Operations might be applied to an inappropriate number of arguments. Behavior might be unspecified in such cases.

- Typed languages
  - Variables are assigned (non-trival) types
  - A type system is the component of a language that keeps track of types
  - Types might or might not appear in the program itself.
  - Languages can be explicitly typed (e.g., Java) or implicitly typed (e.g., ML)
Errors and Soundness

- We’ll concentrate on static type systems:
  - Want to prevent execution errors during program execution.
  - Languages where no program gives rise to an execution error are type-sound.

- Trapped execution errors:
  - Being well-typed does not mean that a program cannot raise an error:
    - division by zero
    - infinite loops that cause a memory segmentation fault.
    - dereferencing an invalid address (in languages with pointer types)
  - Want to ensure that when an execution error does occur, it is detected, and computation stops immediately.
Good Behavior

- For a given language, we designate a set of forbidden errors:
  - All untrapped errors
  - Some trapped errors as well
    - e.g., dereferencing an undefined memory location.
  - A program fragment that does not induce any forbidden errors has good behavior.
  - A language where all legal programs have good behavior is strongly checked.
    - No untrapped errors occur.
    - The programmer is responsible for avoiding some or all trapped errors.
Formalizing Type Systems

A multi-step process:
- Define syntax
  - Expressions (programs)
  - Types
  - Fix binding and scoping issues
- Static semantics (typing rules)
  - Define typing judgments and derivations
- Dynamic semantics (operational semantics)
  - Define evaluation judgments and derivations
- Type soundness
  - Relate static and dynamic semantics
  - Would like to prove that “well-typed programs do not go wrong”
    - Progress: a well-typed term is not stuck
      - It is either a value or it can take a step according to the evaluation rules
    - Preservation: if a well-typed term reduces to another term, the resulting term is also well-typed.
Outline

- Begin with a set of terms, a set of values and an evaluation relation
- Define a set of types classifying values according to their “shape”
  - Intuitively, think of a type as a representative for a (potentially infinite) set of values
- Define a typing relation $t : T$ that classifies terms according to the shape of values (i.e., type) that result from evaluating them
- Check the relation is sound:
  - if $t : T$ and $t \rightarrow^* v$, then $v : T$
  - if $t : T$ then $t$ is not stuck
Typing Judgments

- A judgment is a statement $J$ about certain formal entities
- It may be valid (universally true): $\Gamma \vdash J$
- It may be provable: $\Gamma \vdash J$
- A common form of a typing judgment:
  - $\Gamma \vdash e : \tau$
  - $\Gamma$ is a set of type assignments for the free variables of $e$.
  - Defined by the grammar: $\Gamma ::= . \mid \Gamma, x : \tau$
  - Type assignments for variables not free in $e$ are not relevant:
    - $\{x : \text{int}, y : \text{int}\} \vdash x + y : \text{int}$
Typed Arithmetic

For now, we will ignore variables, and consider expressions built from constants and simple constructors and operations

\[ t ::= \]
\[ \text{true} \mid \text{false} \mid \]
\[ \text{if } t \text{ then } t \text{ else } t \mid \]
\[ 0 \mid \]
\[ \text{succ } t \mid \text{pred } t \mid \text{iszero } t \]

\[ v ::= \]
\[ \text{true} \mid \text{false} \mid \text{nv} \]

\[ \text{nv ::=} \]
\[ 0 \mid \text{succ } \text{nv} \]
Evaluation Rules (Conditionals)

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 $t_3$
Evaluation Rules (Numerals)

\[
\begin{align*}
t_1 & \rightarrow t_1' \\
\text{succ } t_1 & \rightarrow \text{succ } t_1' \\
\text{pred } 0 & \rightarrow 0 \\
\text{pred } (\text{succ } n) & \rightarrow n \\
\text{pred } t_1 & \rightarrow \text{pred } t_1' \\
iszero 0 & \rightarrow \text{true} \\
iszero (\text{succ } n) & \rightarrow \text{false} \\
iszero t_1 & \rightarrow \text{iszero } t_1'
\end{align*}
\]
Types

- In this language, values have two possible shapes:
  - \( T ::= \text{Bool} \mid \text{Nat} \)

- Typing Rules

\[
\begin{align*}
\text{true} &: \text{Bool} \\
\text{false} &: \text{Bool} \\
\text{if } t_1 \text{ then } t_2 \text{ else } t_3 &: T \\
\hline
\end{align*}
\]
Typing Rules (Continued)

\[ 0 : \text{Nat} \]

\[ \frac{t_1 : \text{Nat}}{\text{succ } t_1 : \text{Nat}} \]

\[ \frac{t_1 : \text{Nat}}{\text{pred } t_1 : \text{Nat}} \]

\[ \frac{t_1 : \text{Nat}}{\text{iszero } t_1 : \text{Nat}} \]
Imprecision of Typing

Type systems are a form of static analysis

- They attempt to approximate the actual values denoted by expressions at runtime
- They are an approximation because the values they compute (i.e., types) do not require actually executing the program
- For correctness, they are necessarily conservative:

\[
\begin{align*}
t_1 & : \text{Bool}, t_2 : T, t_3 : T \\
\text{if } t_1 \text{ then } t_2 \text{ else } t_3 & : T
\end{align*}
\]

Using this rule we cannot assign a type to the expression:

if true then 0 else false

even though the term will always evaluate to a number
Type Safety

- The safety (or soundness) of a type system can be expressed by two properties:
  - **Progress**: A well-typed term is not stuck:
    - if \( t : T \) then \( t \) is either a value, or there exists a \( t' \) such that \( t \rightarrow t' \)
  - **Preservation**: Types are preserved by the evaluation function:
    - if \( t : T \) and \( t \rightarrow t' \) then \( t' : T \)
Derivations

A typing derivation is a derivation of a typing judgment, and can be justified by a derivation tree built from instances of the inference rules:

\[
\begin{array}{c}
0 : \text{Nat} \\
\text{iszero } 0 : \text{Bool} \\
\text{if iszero } 0 \text{ then } 0 \text{ else pred } 0 : \text{Nat} \\
\hline
0 : \text{Nat} \\
\hline
\text{pred } 0 : \text{Nat}
\end{array}
\]
Inversion Lemma

1. if true : R then R = Bool
2. if false : R then R = Bool
3. if if \( t_1 \) then \( t_2 \) else \( t_3 \) : R then
   \( t_1 : \text{Bool}, t_2 : R, t_3 : R \)
4. if 0 : R then R = Nat
5. if succ \( t_1 \) : R then R = Nat and \( t_1 : \text{Nat} \)
6. if pred \( t_1 \) : R then R = Nat and \( t_1 : \text{Nat} \)
7. if iszero \( t_1 \) : R then R = Bool and \( t_1 : \text{Nat} \)
Typechecking

typeOf(t) =
    if t = true then Bool
    else if t = false then Bool
    else if t = if t\_1 then t\_2 else t\_3 then
        let T\_1 = \text{typeOf}(t\_1)
        T\_2 = \text{typeOf}(t\_2)
        T\_3 = \text{typeOf}(t\_3)
        in if T\_1 = \text{Bool} and T\_2 = T\_3 then T\_2
            else error
        else if t = 0 then Nat
    else ...

Canonical Forms

Lemma:

1. if \( v \) is a value of type \( \text{Bool} \), then \( v \) is either true or false

2. if \( v \) is a value of type \( \text{Nat} \), then \( v \) is a numeric value

Proof: (part 1): values are either true, false, 0, or succ \( \text{nv} \) where \( \text{nv} \) is numeric. First two cases trivial. Last two cannot occur by inversion lemma
Progress

Suppose $t$ is a well-typed term (i.e., $t : T$ for some type $T$). Then, $t$ is either a value or there is a $t'$ such that $t \rightarrow t'$

Proof: By induction on the derivation of $t : T$. The true, false and zero cases are immediate. Consider remaining cases:

(If): $t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3$

$t_1 : \text{Bool}, t_2 : T, t_3 : T$

By IH, $t_1$ is either a value or there is some $t_1'$ such that $t_1 \rightarrow t_1'$; if $t_1$ is a value, then inversion lemma guarantees its type is $\text{Bool}$. If it is a value, then apply IH to $t_2$ and $t_3$. If it is not a value, then $t \rightarrow \text{if } t' \text{ then } t_2 \text{ else } t_3$.

(Succ) $t = \text{succ } t_1$

By IH, $t_1$ is either a value or there is a $t_1'$ such that $t_1 \rightarrow t_1'$. If $t_1$ is a value, then by inversion lemma, its type is $\text{Nat}$ in which case $t$ is $\text{Nat}$. Otherwise, by evaluation rules, $t \rightarrow \text{succ } t_1'$
Preservation

If \( t : T \) and \( t \rightarrow t' \) then \( t' : T \)
Proof: By induction on the derivation of \( t : T \).

(If) if \( t = \) if \( t_1 \) then \( t_2 \) else \( t_3 \)

\( t_1 : \text{Bool}, t_2 : T, t_3 : T \)

Evaluation rules indicate there are three rules with if on the left-hand side:

\( t_1 = \) true/false, \( t_1' = \) true/false

We're finished since \( t_2/t_3 : T \)

\( t_1 \rightarrow t_1', t' = \) if \( t_1' \) then \( t_2 \) else \( t_3 \)

By IH, \( t_1' : \text{Bool} \) and we know by rules for conditionals that \( t_2 : T \) and \( t_3 : T \) and hence \( t' \) has type \( T \)
Next Time

- We’ve seen a type system for simple types (integers and booleans)
- We’ll next consider languages with function types and type environments (simply-typed $\lambda$-calculus)
- Structured types
  - products for tuples and records
  - sums for algebraic datatypes (trees, lists)
- Imperative types
  - references (mutable objects)
  - exceptions
- Recursive types (related to sum types)
- Subtypes
- Afterwards, will discuss second-order systems
  - Polymorphism
  - Abstract types