Lecture 8: Types and Type Rules

Polyvios Pratikakis

Computer Science Department, University of Crete

Type Systems and Programming Languages

Based on slides by Jeff Foster, UMD
The need for types

- Consider the lambda calculus terms:
  - \( \text{false} = \lambda x.\lambda y. x \)
  - \( 0 = \lambda x.\lambda y. x \) (Scott encoding)

- Everything is encoded using functions
  - One can easily misuse combinators
    - \( \text{false} 0 \), or if \( 0 \) then ... , etc...
    - It’s no better than assembly language!
A *type system* is some mechanism for distinguishing good programs from bad

- Good programs are *well typed*
- Bad programs are ill typed or not typeable

**Examples:**

- $0 + 1$ is well typed
- $\text{false} + 0$ is ill typed: booleans cannot be added to numbers
- $1 + (\text{if} \; \text{true} \; \text{then} \; 0 \; \text{else} \; \text{false})$ is ill typed: cannot add a boolean to an integer

**This time:** types for simple arithmetic (Lecture 4)
A definition

“A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute.”

– Benjamin Pierce, Types and Programming Languages
Recall simple arithmetic

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

\[
    v ::= \text{true} \quad \text{false} \quad \text{nv} \\
    \quad 0 \quad \text{succ} \, \text{nv}
\]
Semantics

- \( \text{iszero } 0 \rightarrow \text{true} \)
- \( \text{iszero } t \rightarrow \text{iszero } t' \)
- \( \nu \text{ is a num. value} \rightarrow \text{false} \)
- \( \text{succ } t \rightarrow \text{succ } t' \)
- \( \text{pred } 0 \rightarrow 0 \)
- \( \text{pred } t \rightarrow \text{pred } t' \)
- \( \nu \text{ is a num. value} \rightarrow \nu \)
- \( \text{if true then } t_1 \text{ else } t_2 \rightarrow t_1 \)
- \( \text{if false then } t_1 \text{ else } t_2 \rightarrow t_2 \)
- \( \text{if } t \text{ then } t_1 \text{ else } t_2 \rightarrow \text{if } t' \text{ then } t_1 \text{ else } t_2 \)
Types: approximation of result

- Classify terms into types:
  - A term $t$ has type $T$: its result will be a boolean/natural
  - Written $t : T$ (sometimes $t \in T$)
  - Computed statically: without running the program
  - Statical typing is conservative: might reject good programs

- For this language we need two types, $T ::= \text{Bool} \mid \text{Nat}$

- Examples:
  - if true then 0 else succ 0 : $\text{Nat}$, always produces a number
  - iszero (suc (pred 0)) : $\text{Bool}$, always produces a boolean
  - But: if true then false else succ 0 does not have a static type
The typing relation

- Define a relation “:" to assign types to terms
- Mathematically, “:" is a partial binary relation between the set $\mathcal{E}$ of all possible programs, and the set $\mathcal{T}$, (here $\{\text{Bool, Nat}\}$) of all possible types
- Can describe this using sets:
  - Language: a set $\mathcal{E}$ of all possible terms
  - Type language: a set $\mathcal{T}$ of all possible types
  - Typing relation: a partial relation “:" $\subseteq \mathcal{E} \times \mathcal{T}$
  - Well-formed terms: a set $\mathcal{WF} \subseteq \mathcal{E}$ of terms that don’t get stuck during evaluation
  - Well-typed terms: a set $\mathcal{WT} \subseteq \mathcal{E}$ of terms that have a type
The typing relation (cont’d)

- When $\mathcal{W} \subseteq \mathcal{WF}$, the type system is sound.
- When $\mathcal{WF} \subseteq \mathcal{W}$, the type system is complete.
- Usually, we can’t have both: undecidable.
- Traditionally, type-systems worry about soundness:
  - i.e: no accepted program can go wrong.
- But might reject some correct programs.
Back to language definitions

- Inductive: the \textit{smallest} set \( \mathcal{E} \) such that
  - \( \{ \text{true, false} \} \in \mathcal{E} \)
  - If \( t_1 \in \mathcal{E} \) then \( \{ \text{succ } t_1, \text{pred } t_1, \text{iszero } t_1 \} \in \mathcal{E} \)
  - etc.

- By inference rules, e.g:

  \[
  \frac{t \in \mathcal{E}}{\text{iszero } t \in \mathcal{E}}
  \]

- By construction:
  - \( S_0 = \emptyset \)
  - \( S_{i+1} = \{ \text{true, false, 0} \} \cup \text{succ } t, \text{pred } t, \text{iszero } t \mid t \in S_i \cup \ldots \)
  - \( \mathcal{E} = \bigcup_i S_i \)
Same thing for typing relation

- **Inductive:** The *smallest* relation \( \triangleright \) such that
  - \( 0 : \text{Nat} \) holds
  - If \( t : \text{Nat} \) holds, then \( \text{succ} \ t : \text{Nat} \) also holds
  - etc.

- **By inference rules:**
  \[
  \frac{t : \text{Nat}}{\text{succ} \ t : \text{Nat}}
  \]

- **By construction:**
  - \( T_0 = \emptyset \)
  - \( T_{i+1} = \{0 : \text{Nat}\} \cup \{\text{succ} \ t : \text{Nat} | (t : \text{Nat}) \in T_i\} \cup \ldots \)
  - \( \mathcal{T} = \bigcup_i T_i \)
Type system

\[
\begin{align*}
[T-\text{TRUE}] & : \text{true} : Bool \\
[T-\text{FALSE}] & : \text{false} : Bool \\
[T-\text{IF}] & : t_1 : \text{Bool} \quad t_2 : T \quad t_3 : T \\
& \quad \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T \\
[T-\text{ZERO}] & : 0 : \text{Nat} \\
[T-\text{SUCCESSION}] & : t : \text{Nat} \\
& \quad \text{succ } t : \text{Nat} \\
[T-\text{PRECEDE}] & : t : \text{Nat} \\
& \quad \text{pred } t : \text{Nat} \\
[T-\text{ISZERO}] & : t : \text{Nat} \\
& \quad \text{iszero } t : \text{Bool}
\end{align*}
\]
Inversion lemma

- Typing relation is the *smallest* relation produced by the rules
- And is syntax-driven (deterministic)
- So we can invert it (inversion lemma):
  - The only way to type true is \([T-\text{True}]\), with type *Bool*
  - The only way to type false is \([T-\text{False}]\), with type *Bool*
  - If there is a typing if \(t_1\) then \(t_2\) else \(t_3\) : \(T\) then the only way to create it is \([T-\text{If}]\), where \(t_1 : \text{Bool}\), \(t_2 : T\) and \(t_3 : T\)
  - etc, for the other syntactic forms
- Proof follows from the definition of typing
- Makes inference rules go backwards:
  - Given the conclusion, the premises must have been true (there is no other way to reach that conclusion)
- Practically, it describes the algorithm to construct a typing
In OCaml

- Grammar (Lec. 4):

  ```ocaml
type term =
  | True
  | False
  | If of term ∗ term ∗ term
  | Zero
  | Succ of term
  | Pred of term
  | IsZero of term
  ```

- Type language:

  ```ocaml
type typ = TNat | TBool
  ```
Type checking

let rec typecheck : term -> typ = function
  | True | False -> TBool
  | If (t1, t2, t3) when typecheck t1 = TBool ->
    let typ2 = typecheck t2 in
    let typ3 = typecheck t3 in
    if (typ2 = typ3) then typ2
    else failwith "type error"
  | Zero -> TNat
  | Succ t | Pred t when (typecheck t) = TNat -> TNat
  | IsZero t when (typecheck t) = TNat -> TBool
  | _ -> failwith "type error"
Progress theorem

- If $t : T$ then either $t$ is a value, or there exists $t'$ such that $t \rightarrow t'$
- Proof by induction on $t$
  - Base cases (simple values): true, false, 0, trivially true
  - Inductive cases: assume sub-terms are either values or can step
    - Case succ $t$: if $t$ is a value then succ $t$ is a value, otherwise $t \rightarrow t'$, therefore succ $t \rightarrow$ succ $t'$ using the fourth semantic rule
    - Case pred $t$: from inversion, we know $t : \text{Nat}$. If $t$ is a value it cannot be true or false. So, we can always take a step from pred 0 or pred (succ $v$). If $t$ is not a value, $t$ takes a step, and pred $t \rightarrow$ pred $t'$
    - ...similarly for the other cases
Preservation theorem

- If $t : T$ and $t \rightarrow t'$ then $t' : T$
- Proof by induction on $t \rightarrow t'$ (each semantic rule)
  - First rule (base case) iszero $0 \rightarrow true$: From inversion lemma on iszero $0 : T$, we get that its type must be $Bool$, which is also the type of true from [T-TRUE]
  - Second rule (inductive case) iszero $t \rightarrow$ iszero $t'$: From inversion lemma on iszero $t : T$ we get $T = Bool$ and also $t : Nat$. From induction hypothesis we have $t \rightarrow t'$. Apply inductively on $t : Nat$ and $t \rightarrow t'$, to get $t' : Nat$. Then iszero $t' : Bool$ follows from [T-ISZERO]
  - Similarly for other base and inductive cases
Soundness

So far:

- Progress: If $t : T$, then either $t$ is a value, or there exists $t'$ such that $t \rightarrow t'$
- Preservation: If $t : T$ and $t \rightarrow t'$ then $t' : T$

Putting these together, we get soundness

- If $t : T$ then either there exists a value $v$ such that $t \rightarrow^* v$ or $t$ doesn't terminate

What does this mean?

- “Well-typed programs don’t go wrong”
- Evaluation never gets stuck

This language will always terminate

- Proof by induction on term size (defined in Lec. 4)
- If $t \rightarrow t'$ then $\text{size}(t') < \text{size}(t)$
Next time

- The same, only for \( \lambda \)-calculus
  - The function type
  - What happens with variables?
  - What happens with substitution?