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:
  - false $= \lambda x.\lambda y.x$
  - $0 = \lambda x.\lambda y.x$ (Scott encoding)
- Everything is encoded using functions
  - One can easily misuse combinators
    - false 0, or if 0 then $\ldots$, etc...
  - It's no better than assembly language!

Type system

- 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
  - false $+ 0$ is ill typed: booleans cannot be added to numbers
  - $1 + ($if true then 0 else 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

$$
\begin{aligned}
t \quad ::= \quad & \text{true} \\
\mid \quad & \text{false} \\
\mid \quad & 0 \\
\mid \quad & \text{succ } t \\
\mid \quad & \text{pred } t \\
\mid \quad & \text{iszero } t \\
\mid \quad & \text{if } t \text{ then } t \text{ else } t
\end{aligned}
$$

$$
\begin{aligned}
v \quad ::= \quad & \text{true} \\
\mid \quad & \text{false} \\
\mid \quad & nv \\
\\
nv \quad ::= \quad & 0 \\
\mid \quad & \text{succ } nv
\end{aligned}
$$

Semantics

$$\frac{}{\text{iszero } 0 \rightarrow \text{true}} \qquad \frac{t \rightarrow t'}{\text{iszero } t \rightarrow \text{iszero } t'}$$

$$\frac{v \text{ is a num. value}}{\text{iszero } (\text{succ } v) \rightarrow \text{false}} \qquad \frac{t \rightarrow t'}{\text{succ } t \rightarrow \text{succ } t'}$$

$$\frac{}{\text{pred } 0 \rightarrow 0} \qquad \frac{t \rightarrow t'}{\text{pred } t \rightarrow \text{pred } t'} \qquad \frac{v \text{ is a num. value}}{\text{pred } (\text{succ } v) \rightarrow v}$$

$$\frac{}{\text{if true then } t_1 \text{ else } t_2 \rightarrow t_1} \qquad \frac{}{\text{if false then } t_1 \text{ else } t_2 \rightarrow t_2}$$

$$\frac{t \rightarrow t'}{\text{if } t \text{ then } t_1 \text{ else } t_2 \rightarrow \text{if } t' \text{ then } t_1 \text{ else } t_2}$$

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, $T ::= Bool \mid Nat$
- Examples:
  - if true then 0 else succ 0 : Nat, always produces a number
  - iszero (succ (pred 0)) : $Bool$, 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 $\{Bool, Nat\}$) 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 $\mathcal{WT} \subseteq \mathcal{WF}$, the type system is *sound*
- When $\mathcal{WF} \subseteq \mathcal{WT}$, 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 *smallest* set $\mathcal{E}$ such that
  - $\{\text{true}, \text{false}\} \in \mathcal{E}$
  - 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:
  - $S_0 = \emptyset$
  - $S_{i+1} = \{\text{true}, \text{false}, 0\} \cup \text{succ } t, \text{pred } t, \text{iszero } t \mid t \in S_i \cup \ldots$
  - $\mathcal{E} = \bigcup_i S_i$

Same thing for typing relation

- Inductive: The *smallest* relation : such that
  - $0 : Nat$ holds
  - If $t : Nat$ holds, then succ $t : Nat$ also holds
  - etc.
- By inference rules:

$$\frac{t : Nat}{\text{succ } t : Nat}$$

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

Type system

$$[\textsc{T-True}]\ \frac{}{\mathsf{true} : Bool} \qquad [\textsc{T-False}]\ \frac{}{\mathsf{false} : Bool}$$

$$[\textsc{T-If}]\ \frac{t_1 : Bool \qquad t_2 : T \qquad t_3 : T}{\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 : T}$$

$$[\textsc{T-Zero}]\ \frac{}{0 : Nat} \qquad [\textsc{T-Succ}]\ \frac{t : Nat}{\mathsf{succ}\ t : Nat}$$

$$[\textsc{T-Pred}]\ \frac{t : Nat}{\mathsf{pred}\ t : Nat} \qquad [\textsc{T-IsZero}]\ \frac{t : Nat}{\mathsf{iszero}\ t : Bool}$$

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-True], with type $Bool$
  - The only way to type false is [T-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-If], where $t_1 : Bool$, $t_2 : T$ 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):

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

- Type language:

```
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 $t \to t'$*
- Proof by induction on t
  - Base cases (simple values): $\text{true}, \text{false}, 0$, 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 $t \to t'$, therefore succ $t \to$ succ t' using the fourth semantic rule
    - ⋆ Case pred t: from inversion, we know $t : Nat$. 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 $t \to$ pred t'
    - ⋆ ...similarly for the other cases

Preservation theorem

- *If $t : T$ and $t \to t'$ then $t' : T$*
- Proof by induction on $t \to t'$ (each semantic rule)
  - First rule (base case) iszero $0 \to$ 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 $t \to$ iszero t': From inversion lemma on iszero $t : T$ we get $T = Bool$ and also $t : Nat$. From induction hypothesis we have $t \to t'$. Apply inductively on $t : Nat$ and $t \to t'$, 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 $t \to t'$
  - Preservation: If $t : T$ and $t \to t'$ then $t' : T$
- Putting these together, we get *soundness*
  - *If $t : T$ then either there exists a value v such that $t \to^* v$ 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 $t \to t'$ then $size(t') < size(t)$

Next time

- The same, only for λ-calculus
  - The function type
  - What happens with variables?
  - What happens with substitution?