Types and Programming Languages

Lecture 4. Types, the simply typed $\lambda$-calculus

Xiaojuan Cai

cxj@sjtu.edu.cn

BASICS Lab, Shanghai Jiao Tong University

Spring, 2016
Outline

Typed arithmetic expressions
  Typing relation
  Safety = Progress + Preservation

Simply typed $\lambda$-calculus
  Function types
Q: Why bother doing proofs about programming languages? They are almost always boring if the definitions are right.

A: The definitions are almost always wrong.
Arithmetic expressions

\[ t ::= \]
\[ \text{true} \quad \text{constant true} \]
\[ \text{false} \quad \text{constant false} \]
\[ \text{if } t \text{ then } t \text{ else } t \quad \text{conditional} \]
\[ 0 \quad \text{constant zero} \]
\[ \text{succ } t \quad \text{successor} \]
\[ \text{pred } t \quad \text{predecessor} \]
\[ \text{iszero } t \quad \text{zero test} \]

\[ v ::= \]
\[ \text{true} \quad \text{true value} \]
\[ \text{false} \quad \text{false value} \]
\[ \text{nv} \quad \text{numeric value} \]

\[ \text{nv} ::= \]
\[ 0 \quad \text{zero value} \]
\[ \text{succ } \text{nv} \quad \text{successor value} \]
Recall that evaluating a term can either result in a value or else get stuck at some stage, by reaching a term like `pred false`.

In fact, we can tell *stuck terms* without actually evaluating it.

Coming soon: If a term is well typed, i.e., it has some type $T$, then it never get stuck (never goes wrong).
Typing relation

The typing relation for arithmetic expressions, written \( t : T \), is defined by a set of inference rules assigning types to terms.

\[
T ::= \text{types} \\
\quad \text{Bool} \quad \text{type of booleans} \\
\quad \text{Nat} \quad \text{type of natural numbers}
\]

Typing rules:

\[
\begin{align*}
\text{T-True} \quad & \text{true} : \text{Bool} \\
\text{T-False} \quad & \text{false} : \text{Bool} \\
\text{T-Zero} \quad & 0 : \text{Nat} \\
\text{T-Succ} \quad & \text{succ } t_1 : \text{Nat} \\
\text{T-Pred} \quad & \text{pred } t_1 : \text{Nat} \\
\text{T-If} \quad & \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T \\
\text{T-IsZero} \quad & \text{iszero } t_1 : \text{Bool}
\end{align*}
\]
Uniqueness of types

When reasoning about the typing relation, we will often inverse the typing relation.

**Lemma 8.2.2:**

1. If `true : R` or `false : R`, then `R = Bool`;
2. If `if t_1 then t_2 else t_3 : R`, then `t_1 : Bool`, `t_2 : R`, and `t_3 : R`.
3. If `0 : R`, or `succ t_1 : R`, or `pred t_1 : R`, then `R = Nat` and `t_1 : Nat`.
4. If `iszero t_1 : R`, then `R = Bool` and `t_1 : Nat`.

**Theorem 8.2.4 [Uniqueness of types]:** Each term `t` has at most one type. Above theorem does not hold for languages with subtyping rules.
The most basic property of type system

\[ \text{Safety} = \text{Progress} + \text{Preservation} \]

- **Progress**: A well-typed term is not stuck (either it 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, then the resulting term is also well typed.
Lemma 8.3.1 [Canonical forms]:

- If \( v \) is a value of type \( \text{Bool} \), then \( v \) is either true or false.
- If \( v \) is a value of type \( \text{Nat} \), then \( v \) is a numeric value according to the grammar.

Theorem 8.3.2 [Progress]: Suppose \( t \) is a well-typed term (that is, \( t : T \) for some \( T \)). Then either \( t \) is a value or else there is some \( t' \) with \( t \rightarrow t' \).

Proof. By induction on a derivation of \( t : T \).
Theorem 8.3.3 [Preservation]: If $t : T$ and $t \rightarrow t'$, then $t' : T$.

Proof. Either by induction on a derivation of $t : T$, or by induction on a derivation of $t \rightarrow t'$. 
Outline

Typed arithmetic expressions
  Typing relation
  Safety = Progress + Preservation

Simply typed $\lambda$-calculus
  Function types
Add types to $\lambda$-calculus

Coming soon: A typing relation for variables, abstractions, and applications that

- maintain type safety: satisfy the type progress and preservation;
- are not to conservative: they should assign types to most of the programs we actually care about writing.

Turing completeness of $\lambda$-calculus implies that there is no hope of giving an exact type analysis for these primitives. For example:

\[
\text{if } \langle \text{long and tricky computation} \rangle \text{ then true else } (\lambda x.x)\\
\]
Arrow type

For a function,

1. we care about the types of both arguments and results:

   arrow type $T \rightarrow T$

   Note the difference between $T \rightarrow T \rightarrow T$ and $(T \rightarrow T) \rightarrow T$

2. the type of an abstraction relies on the type of argument, e.g.

   $\lambda x.x : \text{Bool} \rightarrow \text{Bool}$ or $\lambda x.x : \text{Nat} \rightarrow \text{Nat}$

3. Hence, the typing relation on abstractions should be written as

   $\lambda x : T_1 . t_2 : T_1 \rightarrow T_2$

   But how do we derive $T_2$? We assume $x : T_1$!!! So we need an environment (context) for our typing relation:

   $\Gamma \vdash t : T$
Pure simply typed \(\lambda\)-calculus (\(\lambda\to\))

**Terms**

\[ t ::= x \mid \lambda x : T . t \mid t \; t \]

**Values**

\[ v ::= \lambda x : T . t \]

**Types**

\[ T ::= T \to T \]

**Contexts**

\[ \Gamma ::= \emptyset \mid \Gamma, x : T \]

**Typing**

\[
\begin{align*}
T-\text{VAR} & : x : T \in \Gamma \quad \Rightarrow \quad \Gamma \vdash x : T \\
T-\text{ABS} & : \Gamma, x : T_1 \vdash t_2 : T_2 \quad \Rightarrow \quad \Gamma \vdash \lambda x : T_1 . t_2 : T_1 \to T_2 \\
T-\text{APP} & : \Gamma \vdash t_1 : T_1 \to T_2 \quad \Gamma \vdash t_2 : T_1 \quad \Rightarrow \quad \Gamma \vdash t_1 \; t_2 : T_2
\end{align*}
\]

**Quiz:**
1. Please draw the type derivation tree of the term 
   \((\lambda x : \text{Bool} \to \text{Nat}.x \; \text{true})(\lambda x : \text{Bool}.\text{if} \; x \; \text{then} \; 0 \; \text{else} \; (\text{succ} \; 0))\).
2. What about this term \(\lambda x : \text{Bool}.x \; x\)?
Properties of typing

Lemma 9.3.1 [Inversion of the Typing Relation]:

1. If $\Gamma \vdash x : R$, then $x : R \in \Gamma$.

2. If $\Gamma \vdash \lambda x : T_1 . t_2 : R$, then $R = T_1 \rightarrow R_2$ for some $R_2$ with $\Gamma, x : T_1 \vdash t_2 : R_2$.

3. If $\Gamma \vdash t_1 t_2 : R$, then there is some type $T_1$ such that $t_1 : T_1 \rightarrow R$ and $\Gamma \vdash t_2 : T_1$.

4. for booleans \cdots

Theorem 9.3.3 [Uniqueness of types]: In a given typing context $\Gamma$, a term $t$ (with free variables all in the domain of $\Gamma$) has at most one type.
Lemma 9.3.4 [Canonical forms]:

- If $v$ is a value of type $\text{Bool}$, then $v$ is either true or false.
- If $v$ is a value of type $T_1 \rightarrow T_2$, then $v = \lambda x : T_1.t_2$.

Theorem 9.3.5 [Progress]: Suppose $t$ is a closed, well-typed term (that is, $\Gamma \vdash t : T$ for some $T$). Then either $t$ is a value or else there is some $t$ with $t \rightarrow t$. 
Preservation

**Theorem 9.3.9** [Preservation]: If $\Gamma \vdash t : T$ and $t \rightarrow t'$, then $\Gamma \vdash t' : T$.

**Quiz.** Please try to prove above theorem and figure out what lemmas we need.

**Lemma 9.3.6** [Permutation]: If $\Gamma \vdash t : T$ and $\Delta$ is a permutation of $\Gamma$, then $\Delta \vdash t : T$. Moreover, the latter derivation has the same depth as the former.

**Theorem 9.3.7** [Weakening]: If $\Gamma \vdash t : T$ and $x \not\in \text{dom}(\Gamma)$, then $\Gamma, x : S \vdash t : T$. Moreover, the latter derivation has the same depth as the former.

**Theorem 9.3.8** [Preservation of types under substitution]: If $\Gamma, x : S \vdash t : T$ and $\Gamma \vdash s : S$, then $\Gamma \vdash [x \mapsto s]t : T$. 
The Curry-Howard correspondence

Other names for typing rules from a logic view.
The \( \rightarrow \) type constructor comes with typing rules of two kinds:

- an **introduction** rule \((T-\text{ABS})\) describing how elements of the type can be created, and
- an **elimination** rule \((T-\text{APP})\) describing how elements of the type can be used.

### Curry-Howard Correspondence, or isomorphism

<table>
<thead>
<tr>
<th><strong>Logic</strong></th>
<th><strong>Programming Languages</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>proposition</td>
<td>types</td>
</tr>
<tr>
<td>proposition ( P \supset Q )</td>
<td>type ( P \rightarrow Q )</td>
</tr>
<tr>
<td>proposition ( P \land Q )</td>
<td>type ( P \times Q )</td>
</tr>
<tr>
<td>proof of proposition ( P )</td>
<td>term ( t ) of type ( P )</td>
</tr>
<tr>
<td>proposition ( P ) is provable</td>
<td>type ( P ) is inhabited (by some term)</td>
</tr>
</tbody>
</table>
Erasure and Typability

Type annotations are be used during type checking, and will be erased before evaluation.

**Definition 9.5.1 [Erasure]:** The erasure of a simply typed term $t$ is defined as follows:

\[
\begin{align*}
erase(x) &= x \\
erase(\lambda x : T_1.t_2) &= \lambda x.\,\text{erase}(t_2) \\
erase(t_1 \, t_2) &= \text{erase}(t_1)\text{erase}(t_2)
\end{align*}
\]

**Definition 9.5.3 [Typability]:** A term $m$ in the untyped $\lambda$-calculus is said to be typable in $\lambda \hookrightarrow$ if there are some simply typed term $t$, type $T$, and context $\Gamma$ such that $\text{erase}(t) = m$ and $\Gamma \vdash t : T$. 
Conclusion

- Typing system $\Gamma \vdash t : T$ can remove some terms before they run into stuck states.
- However, it also removes well-behaved terms.
- Type safety = progress + preservation
- For proving safety, some other properties such as canonical forms, uniqueness of type are needed.
- Simply typed $\lambda$-calculus is non-Turing-complete.
Homework

- 8.3.4, 8.3.6, 8.3.7, 9.2.2, 9.2.3, 9.3.2, 9.4.1

Projects. Extend arith with

- Untyped lambda calculus (Chapter 7), due on Apr. 14 (Thursday of Week 8)
- Simple typed lambda calculus (Chapter 10), due on May. 12 (Thursday of Week 12)
- Subtyping (Chapter 17), due on Jun. 9 (Thursday of Week 16)