The Simply Typed Lambda-Calculus

Lambda-calculus with booleans

Operational semantics

<table>
<thead>
<tr>
<th>Condition</th>
<th>Rule</th>
<th>Predecessor</th>
<th>Postdecessor</th>
</tr>
</thead>
<tbody>
<tr>
<td>if <code>t1</code> then <code>t2</code> else <code>t3</code></td>
<td><code>t1</code> → <code>t</code></td>
<td><code>t1</code></td>
<td><code>t2</code></td>
</tr>
<tr>
<td>if <code>false</code> then <code>t2</code> else <code>t3</code></td>
<td><code>t2</code></td>
<td><code>t2</code></td>
<td><code>t3</code></td>
</tr>
<tr>
<td>if <code>true</code> then <code>t2</code> else <code>t3</code></td>
<td><code>t3</code></td>
<td><code>t3</code></td>
<td><code>t2</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
<th>Rule</th>
<th>Predecessor</th>
<th>Postdecessor</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>E-App1</code></td>
<td><code>t2</code></td>
<td><code>t0</code></td>
<td><code>t2</code></td>
</tr>
<tr>
<td><code>E-App2</code></td>
<td><code>t1</code></td>
<td><code>t0</code></td>
<td><code>t1</code></td>
</tr>
<tr>
<td><code>E-AppAbs</code></td>
<td><code>λx.t</code></td>
<td><code>t</code></td>
<td><code>[x:=t]</code></td>
</tr>
</tbody>
</table>

Terms

Values

Conditional

Ax. `t` :: `t`
Typing rules

(L-Var) \[ \frac{ }{ x : T } \]

(L-Type) \[ \frac{ }{ T \rightarrow T } \]

(L-True) \[ \frac{ }{ \text{true} : \text{Bool} } \]

(L-False) \[ \frac{ }{ \text{false} : \text{Bool} } \]

(L-If) \[ \frac{ T \rightarrow \text{false} \rightarrow \text{true} : \text{false} } { x : T \rightarrow \text{false} \rightarrow \text{true} } \]

Type of functions

L-types

\[ L : \text{bool} \]

\[ L = \]
Typing rules

true : Bool

false : Bool

\( t_1 : \text{Bool} \)

\( t_2 : \text{T} \)

\( t_3 : \text{T} \)

\( \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : \text{T} \)

\( x : \text{T} \)

\( t_2 : \text{T} \)

\( x : \text{T} \)

\( t_1 : \text{Bool} \)

\( t_2 : \text{T} \)

\( t_1 : \text{Bool} \)

\( \text{if } x \text{ then } \text{false} \text{ else } x : \text{Bool} \)

Typing Derivations

What derivations justify the following typing statements?

<table>
<thead>
<tr>
<th>Rule</th>
<th>Derivation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{T-Var} )</td>
<td>( \vdash x : \text{T} )</td>
</tr>
<tr>
<td>( \text{T-Line} )</td>
<td>( \vdash \text{true} : \text{Bool} )</td>
</tr>
<tr>
<td>( \text{T-Line} )</td>
<td>( \vdash \text{false} : \text{Bool} )</td>
</tr>
<tr>
<td>( \text{T-Line} )</td>
<td>( \vdash \text{false} : \text{Bool} )</td>
</tr>
<tr>
<td>( \text{T-Line} )</td>
<td>( \vdash \text{true} : \text{Bool} )</td>
</tr>
<tr>
<td>( \text{T-Line} )</td>
<td>( \vdash \text{false} : \text{Bool} )</td>
</tr>
<tr>
<td>( \text{T-Line} )</td>
<td>( \vdash \text{false} : \text{Bool} )</td>
</tr>
<tr>
<td>( \text{T-Line} )</td>
<td>( \vdash \text{true} : \text{Bool} )</td>
</tr>
<tr>
<td>( \text{T-Line} )</td>
<td>( \vdash \text{false} : \text{Bool} )</td>
</tr>
<tr>
<td>( \text{T-Line} )</td>
<td>( \vdash \text{false} : \text{Bool} )</td>
</tr>
<tr>
<td>( \text{T-Line} )</td>
<td>( \vdash \text{true} : \text{Bool} )</td>
</tr>
<tr>
<td>( \text{T-Line} )</td>
<td>( \vdash \text{false} : \text{Bool} )</td>
</tr>
<tr>
<td>( \text{T-Line} )</td>
<td>( \vdash \text{false} : \text{Bool} )</td>
</tr>
<tr>
<td>( \text{T-Line} )</td>
<td>( \vdash \text{true} : \text{Bool} )</td>
</tr>
<tr>
<td>( \text{T-Line} )</td>
<td>( \vdash \text{false} : \text{Bool} )</td>
</tr>
<tr>
<td>( \text{T-Line} )</td>
<td>( \vdash \text{false} : \text{Bool} )</td>
</tr>
<tr>
<td>( \text{T-Line} )</td>
<td>( \vdash \text{true} : \text{Bool} )</td>
</tr>
<tr>
<td>( \text{T-Line} )</td>
<td>( \vdash \text{false} : \text{Bool} )</td>
</tr>
<tr>
<td>( \text{T-Line} )</td>
<td>( \vdash \text{false} : \text{Bool} )</td>
</tr>
<tr>
<td>( \text{T-Line} )</td>
<td>( \vdash \text{true} : \text{Bool} )</td>
</tr>
<tr>
<td>( \text{T-Line} )</td>
<td>( \vdash \text{false} : \text{Bool} )</td>
</tr>
<tr>
<td>( \text{T-Line} )</td>
<td>( \vdash \text{false} : \text{Bool} )</td>
</tr>
<tr>
<td>( \text{T-Line} )</td>
<td>( \vdash \text{true} : \text{Bool} )</td>
</tr>
<tr>
<td>( \text{T-Line} )</td>
<td>( \vdash \text{false} : \text{Bool} )</td>
</tr>
<tr>
<td>( \text{T-Line} )</td>
<td>( \vdash \text{false} : \text{Bool} )</td>
</tr>
<tr>
<td>( \text{T-Line} )</td>
<td>( \vdash \text{true} : \text{Bool} )</td>
</tr>
<tr>
<td>( \text{T-Line} )</td>
<td>( \vdash \text{false} : \text{Bool} )</td>
</tr>
</tbody>
</table>
As before, the fundamental property of the type system we have just defined is soundness with respect to the operational semantics.

1. Progress:
   A closed, well-typed term is not stuck.
   If \( t : T \) and \( \lambda \ x . \ t \to x \), then \( \lambda \ x . \ t \to x \).

2. Preservation:
   Types are preserved by one-step evaluation.
   If \( t : T \) and \( t \to t' \), then \( \lambda \ x . \ t \to \lambda \ x . \ t' \) for some \( T' \).
   If \( t : T \), then \( \lambda \ x . \ t \) is a value or else \( t \to t' \) for some \( T' \).

Progress: \( \lambda \ x . \ t \) closed, well-typed term is not stuck.
Typing rules again (for reference)

Lemma

Canonical Forms

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

2. If \( v \) is a value of type \( T_1 \rightarrow T_2 \), then \( v \) has the form \( x : T_1 . t_2 \).
Lemma: 1. If $v$ is a value of type $\text{Bool}$, then $v$ is either $\text{true}$ or $\text{false}$.

2. If $v$ is a value of type $T_1 \Rightarrow T_2$, then $v$ is either $\text{true} \Rightarrow T_2$ or $\text{false}$.

Theorem: Suppose $t$ is a closed, well-typed term (that is, $t: T$ for some $T$).

Then either $t$ is a value or else there is some $t_0$ with $t \not= t_0$.

Proof: By induction on typing derivations.

The cases for boolean constants and conditions are the same as before.

The variable case is trivial (because $t$ is closed).

The abstraction case is immediate, since abstractions are values.

Consider the case for application, where $t = t_1 t_2$ with $t_1: T_{11} \Rightarrow T_{12}$ and $t_2: T_{11}$.

By the induction hypothesis, either $t_1$ is a value or else it can make a step of evaluation, and likewise $t_2$.

If $t_1$ cannot take a step, then rule $E-\text{App1}$ applies to $t$.

If $t_1$ is a value and $t_2$ cannot take a step, then rule $E-\text{App2}$ applies.

Finally, if both $t_1$ and $t_2$ are values, then the canonical forms lemma tells us that $t_1$ has the form $x: T_{11}.t_{12}$, and so rule $E-\text{AppAbs}$ applies to $t$.

What if $t$ weren't closed?
tells us that if \( t \) has the form \( \lambda \mathbf{x}.t_1.t_2 \), and so the \( E \)-\( \Pi \) rules apply to \( t \).

Consider the abstractions case. Since abstractions are values.

**Theorem:** Suppose \( t \) is a closed, well-typed term (that is, \( \Gamma \vdash t : T \) for some \( T \)).

\[ \begin{array}{c}
\vdash \quad \vdash T_1.T_2 \\
t \cdot \vdash T_1 \\
\end{array} \]

Consider the \( \Pi \)-case for application, where \( t = t_1.t_2 \) with \( \Gamma \vdash t_1 : T_1 \) and \( \Gamma \vdash t_2 : T_2 \). The \( \Pi \) case is trivial (because \( \mathbf{e} \) is closed).

By induction on \( \Pi \)-typing. The case for boolean constants and conditions are the same as before. 

The \( \Pi \) case is trivial (because \( \mathbf{e} \) is closed). The \( \Pi \) case is immediate since abstractions are values.

**Proof:** Induction on \( \Pi \)-typing. The case for boolean constants and conditions are the same as before. 

The \( \Pi \) case is trivial (because \( \mathbf{e} \) is closed). 

By induction on \( \Pi \)-typing. The case for boolean constants and conditions are the same as before. 

The \( \Pi \) case is trivial (because \( \mathbf{e} \) is closed).

By induction on \( \Pi \)-typing. The case for boolean constants and conditions are the same as before. 

The \( \Pi \) case is trivial (because \( \mathbf{e} \) is closed).
Progress

Theorem: 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 \( t \) is some \( \lambda x : T_1. t' \) with \( \Gamma \vdash t : T_1 \).

Proof: Suppose \( t \) is a closed, well-typed term (that is, \( \Gamma \vdash t : T \) for some \( T \)).

Proving Preservation

Theorem: If \( \Gamma \vdash t : T \) and \( t \not\rightarrow t_0 \), then \( \Gamma \vdash t_0 : T \).

Proof: By induction on typing derivations.

\[ \{ \text{Which case is the hard one?} \} \]

Case \( T \text{-App} \):

Given \( t = t_1 t_2 \) \( \Gamma \vdash t_1 : T_1 \not\rightarrow T_12 \) and \( \Gamma \vdash t_2 : T_1 \).

Show \( \Gamma \vdash t_0 : T_12 \).

By the inversion lemma for evaluation, there are three subcases...

Subcase: \( \Gamma \vdash t_1 : T_1 \) and \( t_2 \) is a value \( v \).

\[ t_0 = \lambda x : T_11. t_12 \]

Uhoh.
The Substitution Lemma

**Lemma:** Types are preserved under substitution.

\[ \text{If } \forall x:S. t : T \text{ and } \forall x:S. s : T, \text{ then } \forall x:S. \[x \mapsto s\] t : T. \]

**Proof:** By induction on typing derivations.

**Subcase:** \[ t = \text{ a value } v \]

**Subcase:** \[ t = \text{ App } \]

By the inversion lemma for evaluation, there are three subcases:

**Subcase:** \[ t = \text{ App } \]

By induction on typing derivations, there are three subcases:

**Subcase:** \[ t = \text{ App } \]

By induction on typing derivations.
The Substitution Lemma

Lemma: Types are preserved under substitution.

If \( x : S \) \( \vdash t : T \) and \( \vdash s : S \), then \( \vdash [x \leftarrow s] t : T \).

Proof: ...