The Simply Typed Lambda Calculus

Ralf Lämmel

Resources: The slides of this lecture were derived from [Järvi], with permission of the original author, by copy & paste or by selection, annotation, or rewording. [Järvi] is in turn based on [Pierce] as the underlying textbook.

[Järvi] Slides by J. Järvi: “Programming Languages”, CPSC 604 @ TAMU (2009)
Towards typed lambda calculus

- Now suppose you want to distinguish values of different types:
  - Booleans
  - Numbers
  - Functions on Booleans
  - Functions on functions on Booleans
  - Products, sums of ...
  - ...
- These types need to be ...
  - specified in the program, and
  - checked to be correct.
Setting up the simply typed lambda calculus

• Define syntax simple types and function types.
• Extend lambda abstractions for explicit types.
• Revise reduction semantics.
• Define typing rules.
• Establish type safety.
• Consider extensions.
Revised lambda abstraction

• Lambda abstractions are annotated with types:
  \( \lambda x : T.t \)

• Grammar of types:
  \[ T ::= \text{bool} \]
  \[ \text{nat} \]
  \[ T \rightarrow T \]

We only consider these simple types here for simplicity.
Examples

• What are the types of these terms?

  ✦ \( \lambda x : \text{bool}. x \)

  ✦ \( \lambda f : \text{bool} \rightarrow \text{bool}. f \, x \)

• Here are the same terms for the untyped calculus:

  ✦ \( \lambda x \, .x \)

  ✦ \( \lambda f.f \, x \)

Note that lambda variables are typed explicitly.
Meaningless terms

• Some terms diverge.

• Some applications are ill-typed, e.g.:

\[ (\forall f : \text{bool} \rightarrow \text{bool}.f \, x) \, \text{true} \]

• Goal: a type system to reject ill-typed terms.
Typing relation with context

\[ \Gamma \vdash t : T \quad \text{Term } t \text{ has type } T \text{ in the typing context } \Gamma \]

- A typing context is a sequence of bindings.
- Each binding is a variable-type pair, e.g.: \( x : T \).
- Contexts are composed as in \( \Gamma, x : T \).
- All variable names are distinct for a given \( \Gamma \).
- \( \Gamma \) can be omitted if it is empty.
- \( \Gamma \) can be empty for closed terms.
Typing rules for simply-typed lambda calculus

- $x, y, z, f, g$ range over variables
- $s, t, u$ range over terms
- $S, T, U$ range over types

\[
\begin{align*}
\text{T-Variable} & \quad \Gamma, x : T \vdash x : T \\
\text{T-Abstraction} & \quad \Gamma, x : T \vdash u : U \\
\text{T-Application} & \quad \Gamma \vdash t : U \rightarrow T \quad \Gamma \vdash u : U \\
& \quad \Gamma \vdash t\ u : T
\end{align*}
\]
Rules for bool

T-True
\[ \vdash \text{true} : \text{bool} \]

T-False
\[ \vdash \text{false} : \text{bool} \]

These typing rules illustrate one option to add specific types and their operations to a basic lambda calculus. Basically, we need to add one rule per operation.
Typing derivations

Construct derivations as proofs of terms having a certain type.

\[
\begin{align*}
\Gamma & \vdash f : \text{bool} \rightarrow \text{bool} \\
\Gamma & \vdash f : \text{bool} \rightarrow \text{bool} \\
\Gamma & \vdash f \text{ false} : \text{bool} \\
\Gamma & \vdash (\lambda f : \text{bool} \rightarrow \text{bool}. f \text{ false}) : (\text{bool} \rightarrow \text{bool}) \rightarrow \text{bool} \\
\Gamma & \vdash (\lambda f : \text{bool} \rightarrow \text{bool}. f \text{ false}) \lambda g : \text{bool}. g : \text{bool}
\end{align*}
\]

\[
\begin{align*}
\text{T-Variable} & \quad \text{T-Abstraction} \\
\Gamma & \vdash x : T \quad \Gamma, x : T \vdash u : U \\
\Gamma & \vdash x : T \quad \Gamma \vdash \lambda x : T. u : T \rightarrow U \\
\text{T-Application} & \\
\Gamma & \vdash t : U \rightarrow T \quad \Gamma \vdash u : U \\
\Gamma & \vdash t \ u : T
\end{align*}
\]
Evaluation rules

- Syntax (terms, values, types)
  \[ t ::= x \mid v \mid t \ t \]
  \[ v ::= \lambda x : T . t \mid \text{true} \mid \text{false} \]
  \[ T ::= \text{bool} \mid T \rightarrow T \]

- Evaluation rules
  \[
  \begin{align*}
  & t_1 \rightarrow t_1' \quad t \rightarrow t' \\
  & t_1 \ t_2 \rightarrow t_1' \ t_2 \\
  & (\lambda x : T . t) \ v \rightarrow [v/x]t
  \end{align*}
  \]

Evaluation rules do not bother with types.
Type safety = progress + preservation

**Progress:** If $t$ is a closed, well-typed term, then either $t$ is a value, or there exists some $u$, such that $t \rightarrow u$.

**Preservation:** If $\Gamma \vdash t : T$ and $t \rightarrow u$, then $\Gamma \vdash u : T$

Requires several trivial lemmas (properties) that are omitted here.
A few extensions

• **Recursion** (fixed point combinator)

• **Unit type and sequencing** (for effects eventually)

• **Type annotation** (for documentation, abstraction)

• **Pairs** (as a simple form of type construction)

• **Lists** (another example of type construction)

• **Records** (as a first step towards objects)
Recursion

- A fixed point combinator is definable in the untyped calculus.
- It is not definable in the simply typed version.
  - A special combinator is added to the formal system.
  - Alternatively, a more powerful type system is needed.
Recursion in the presence of types

• Self application is not typeable:
  \( \lambda x : ? . x x \)

• \( Y \) is not typeable either.

• Solution: add a primitive \( \text{fix} \).
  \( t ::= \ldots | \text{fix} \ t \)

Typing rule
\[
\Gamma \vdash t : T \rightarrow T \\
\Gamma \vdash \text{fix} \ t : T
\]

Evaluation rules
\[
\begin{align*}
t & \rightarrow t' \\
\text{fix} \ t & \rightarrow \text{fix} \ t'
\end{align*}
\]
\[
\text{fix} \ (\lambda x : T. t) \rightarrow [(\text{fix} \ (\lambda x : T. t))/x] \ t
\]
Illustration of fix

is\textit{even} : \textit{nat} \rightarrow \textit{bool}
is\textit{even} = \textit{fix} \ g

g : (\textit{nat} \rightarrow \textit{bool}) \rightarrow \textit{nat} \rightarrow \textit{bool}
g = \lambda \ e:\textit{nat} \rightarrow \textit{bool}. \lambda \ x:\textit{nat}. \begin{align*}
&\text{if iszero } x \text{ then true} \\
&\quad \text{else if iszero } (\text{pred } x) \text{ then false} \\
&\quad \text{else } e (\text{pred } (\text{pred } x))
\end{align*}

\textit{g} is a generator for the is\textit{even} function. Given a function that equates with is\textit{even} for numbers up to \( n \), \( g \) defines an approximation up to \( n + 2 \). fix \( g \) extends this to all \( n \).
Unit type and sequencing

- New syntax: $t ::= \ldots \text{unit} | t; t$
- New value: $v ::= \ldots \text{unit}$
- New type: $T ::= \ldots \text{unit}$
- Typing of unit and sequencing:

  $\Gamma \vdash \text{unit} : \text{unit}$

  $\Gamma \vdash t : \text{unit} \quad \Gamma \vdash u : U$

  $\Gamma \vdash t; u : U$

- Evaluation of sequencing:

  $t \rightarrow u$

  $\frac{t \rightarrow u}{t; s \rightarrow u; s}$

  unit; $u \rightarrow u$
Type annotation (ascription)

• Syntax:

\[ t ::= \ldots | t \text{ as } T \]

• Typing rule:

\[ \Gamma \vdash t : T \]
\[ \Gamma \vdash t \text{ as } T : T \]

• Evaluation rules:

\[ t \rightarrow u \]
\[ t \text{ as } T \rightarrow u \text{ as } T \]
\[ v \text{ as } T \rightarrow v \]
Pairs

- New syntax: \( t ::= \ldots \{t, t\} \mid t.1 \mid t.2 \)
- Typing of pairs:

\[
\begin{align*}
\Gamma \vdash t : T & \quad \Gamma \vdash u : U \\
\Gamma \vdash \{t, u\} : T \times U
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash t : T \times U & \quad \Gamma \vdash t : T \times U \\
\Gamma \vdash t.1 : T & \quad \Gamma \vdash t.2 : U
\end{align*}
\]

- Evaluation rules:

\[
\begin{align*}
\{v_1, v_2\}.1 & \to v_1 & \{v_1, v_2\}.2 & \to v_2 \\
\{v, u\} & \to \{v, u'\} & t & \to t' & \{t, u\} & \to \{t', u\} \\
\end{align*}
\]

\[
\begin{align*}
\{v, u\} & \to \{v, u'\} & t.1 & \to t'.1 & t.2 & \to t'.2
\end{align*}
\]
Lists

- New type: ... | List T
- New congruence rules, e.g.:
  \[
  \frac{t_1 \rightarrow t'_1}{\text{cons}[T] \ t_1 \ t_2 \rightarrow \text{cons}[T] \ t'_1 \ t_2}
  \]
- New computation rules, e.g.:
  \[
  \text{head}[S] \ (\text{cons}[T] \ v_1 \ v_2) \rightarrow v_1
  \]
- New typing rules, e.g.:
  \[
  \frac{\Gamma \vdash t : \text{List} \ T}{\Gamma \vdash \text{head}[T] \ t : T}
  \]
Records

• Pairs generalize to tuples.
• Tuples further generalize to records.
• Records generalize to extensible records.
• Extensible records generalize to objects.

We use the syntax:

{age=44, name="Smith"}       // record value
{age=44, name="Smith"}.name    // field access

and write the types as:

{age=44, name="Smith"} : {age:Int, name:String}
Records

- New syntax: \( t ::= \ldots \{ l_i = t_i^{i \in 1\ldots n} \} \mid t.l \)
- New values: \( v ::= \ldots \{ l_i = v_i^{i \in 1\ldots n} \} \)
- New types: \( T ::= \ldots \{ l_i : T_i^{i \in 1\ldots n} \} \)
- Typing of records:
  
  for each \( i, \Gamma \vdash t_i : T_i \)
  
  \[ \Gamma \vdash \{ l_i = t_i^{i \in 1\ldots n} \} : \{ l_i : T_i^{i \in 1\ldots n} \} \]

  \[ \Gamma \vdash t : \{ l_i : T_i^{i \in 1\ldots n} \} \]

  \[ \Gamma \vdash t.l : T_j \]

- Evaluation rules:

  \[ \{ l_i = v_i^{i \in 1\ldots n} \}.l_j \rightarrow v_j \]

  \[ t \rightarrow t' \]

  \[ t.l \rightarrow t'.l \]

  \[ t_j \rightarrow t'_j \]

  \[ \{ l_i = v_i^{i \in 1\ldots j-1}, l_j = t_j, l_k = t_k^{k \in j+1\ldots n} \} \]

  \[ \rightarrow \{ l_i = v_i^{i \in 1\ldots j-1}, l_j = t'_j, l_k = t_k^{k \in j+1\ldots n} \} \]
• **Summary**: The typed lambda calculus
  ✦ Typing relation carries argument for context.
  ✦ Many forms of types can be added modularly.
  ✦ Recursion requires built-in Y combinator.

• **Prepping**: “Types and Programming Languages”
  ✦ Chapters 7 and 11

• **Outlook**:
  ✦ Lambda calculi with polymorphism?
  ✦ Functional programming with Haskell!