Simply Typed $\lambda$-calculus

Lecture 1

Jeremy Dawson

The Australian National University

Semester 2, 2017
A Brief History of Type Theory

- First developed by Bertrand Russell (around 1903), to avoid certain paradoxes in the foundations of math, e.g., Russell’s paradox.
- A hierarchy of *types* introduced to solve this. Typically, a class of all sets is of a different type than a set, etc.
- Formalised by Alonzo Church in the 1930’s in his *simple type theory*, which introduces types to (untyped) \( \lambda \)-calculus.
- The full simple type theory deals with logical concepts as well (also known as higher-order logic).
- We shall deal only with the functional part, i.e., we only consider the \( \beta \)-equality part of Church’s simple theory of types.
Types in computation

- Types are used to restrict the *behaviours* of a program.
- Maxim: Well-typed programs don’t go *wrong*.
- In the case of $\lambda$-calculus, types can be used to enforce *termination*.
- Think of it as a light-weight theorem proving: types correspond to the property we want the program to satisfy.
Outline of the course

- Lecture 1: Introduction, Church & Curry type systems.
- Lecture 2: The type preservation theorem.
- Lecture 3: Type inference.
- Lecture 4: The normalisation theorem (typed terms always have normal forms); Kripke semantics.
- Lecture 5: Curry-Howard correspondence, relating intuitionistic logic and simply typed lambda calculus.
References

Course materials will be made available on Wattle and/or the COMP4630 website:
Email me at jeremy@cecs.anu.edu.au, phone (612)57778
I am in the RSISE building, bldg 115, Tuesdays, Thursdays and alternate Mondays.
Overview of untyped $\lambda$-calculus

- Syntax:
  \[ M, N ::= \, x \mid M \, N \mid \lambda x.\, M \]

- $\alpha$-equivalence: syntactic equality modulo renaming of bound variables, e.g., $\lambda x\lambda y.\, x \equiv \lambda u\lambda v.\, u$.

- Substitution.
  - Notation: $M[x := N]$. (Other notations exist also).
  - Capture avoiding: no free variables in $N$ got captured in $M$ after substitution: use $\alpha$-equivalence first to avoid the problem
    \[
    (\lambda v.\, u \, v)[u := v] \not\equiv \lambda v.\, v \, v
    \]
    but \( (\lambda v.\, u \, v)[u := v] \equiv \lambda w.\, v \, w \).
  - Substitute for free variables only
    \[
    (\lambda v.\, u \, v)[v := w] \not\equiv \lambda v.\, u \, w
    \]
    but \( (\lambda v.\, u \, v)[v := w] \equiv \lambda v.\, u \, v \).

- Computation via $\beta$-reduction:
  \[
  (\lambda x.\, M)\, N \rightarrow_\beta M[x := N]
  \]
Two type systems

There are two ways to define typed \( \lambda \)-calculus:

- **Curry’s style: type assignment** (also, *implicit typing*)
  - Terms are as untyped \( \lambda \)-calculus.
  - Given a term \( M \), what types can be assigned to \( M \)? There can be zero or many.

- **Church’s style: type annotations** (also, *explicit typing*).
  - Terms are explicitly annotated with type expressions.
  - Given a term \( M \), if it is typeable, it has a unique type.

We shall see that the two systems are essentially equivalent.
Some questions

Some typical questions one asks about a type system:

- **What are the properties of well-typed terms?**
  - **Termination:** A well-typed term has a normal form.
  - **Type preservation:** If $M$ has type $\tau$ and $M \rightarrow^\beta N$, then $N$ has type $\tau$.

- **Type checking:** Given a term $M$ and a type $\tau$, is it decidable to check whether $M$ has type $\tau$?

- **Type inference:** Given a term $M$, is there a type $\tau$ such that $M$ has type $\tau$?

- **Type inhabitant:** Given a type $\tau$, is there a term $M$ such that $M$ has type $\tau$?

We’ll see answers to some of these questions for simple type systems.
Simple types

• Assume an infinite set of type variables. We use $\alpha, \beta, \gamma$ to denote type variables.
• The set $\mathbb{T}$ of simple types is the smallest set such that:
  ▶ Type variables: $\alpha \in \mathbb{T}$, for every type variable $\alpha$.
  ▶ Function types: If $\tau_1, \tau_2 \in \mathbb{T}$, then $\tau_1 \to \tau_2 \in \mathbb{T}$.
• We write $\tau \equiv \tau'$ if $\tau$ and $\tau'$ are syntactically equal.
• Notational convention: we assume $\to$ associate to the right, so

$$\alpha_1 \to \alpha_2 \to \alpha_3$$

means

$$\alpha_1 \to (\alpha_2 \to \alpha_3).$$
Typing statements, type declarations and typing contexts

- When assigning a type to a term (in both Church & Curry type systems), we to assign types to free variables in the term.
- A type declaration is a pair $x : \tau$ of a (term) variable $x$ and a type $\tau$.
- A typing statement is a pair $M : \tau$ of a term and a type (read $M$ has type $\tau$).
- A typing context $\Gamma$ is a finite set of typing declarations

$$\{x_1 : \tau_1, \ldots, x_n : \tau_n\}$$

where $x_1, \ldots, x_n$ are pairwise distinct.

- Typing judgments:

$$\Gamma \vdash M : \tau$$

Read: the statement $M : \tau$ is derivable (in a type system) from the typing context $\Gamma$
The typing judgment

\[ \Gamma \vdash_{\lambda Cu} M : \tau \]

is derivable in \( \lambda Cu \) if \( \Gamma \vdash_{\lambda Cu} M : \tau \) can be produced by the following rules:

- **axiom**, if \((x : \tau) \in \Gamma\)

  \[\Gamma \vdash_{\lambda Cu} x : \tau\]

- **\(\rightarrow\) intro**

  \[\Gamma, x : \tau_1 \vdash_{\lambda Cu} M : \tau_2 \quad \begin{align*} & \Gamma \vdash_{\lambda Cu} (\lambda x. M) : \tau_1 \rightarrow \tau_2 \end{align*} \rightarrow \text{intro}\]

- **\(\rightarrow\) elim**

  \[\begin{align*} & \Gamma \vdash_{\lambda Cu} M : \tau' \rightarrow \tau \quad \Gamma \vdash_{\lambda Cu} N : \tau' \\ & \Gamma \vdash_{\lambda Cu} (M \ N) : \tau \end{align*} \rightarrow \text{elim}\]

Here \( \Gamma, x : \tau_1 \) means \( \Gamma \cup \{x : \tau_1\} \).

Note that in \(\rightarrow\) intro, \(x\) has to be "fresh" w.r.t. \(\Gamma\) (ie, \(x\) not in \(\Gamma\)).
Example of a typing deduction (or typing derivation)

For simplicity, we omit the subscript $\lambda_{Cu}$ in $\vdash \lambda_{Cu}$:

\[
\begin{align*}
  & x : \alpha \to \beta, y : \alpha \vdash x : \alpha \to \beta \\
  & x : \alpha \to \beta, y : \alpha \vdash y : \alpha \\
  & x : \alpha \to \beta, y : \alpha \vdash (x \ y) : \beta \\
  & x : \alpha \to \beta \vdash (\lambda y. x \ y) : \alpha \to \beta \\
  & \vdash (\lambda x. \lambda y. x \ y) : (\alpha \to \beta) \to \alpha \to \beta
\end{align*}
\]

Exercises: prove the following typing judgments.

- $\vdash (\lambda x. x) : \alpha \to \alpha$.
- $\vdash (\lambda x. x) : (\beta \to \beta) \to (\beta \to \beta)$.
- $\vdash \lambda x \lambda y \lambda z. x \ z \ (y \ z) : (\alpha \to \beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma$. 
Example: an untypeable term

The term $\lambda x.x \ x$ is untypeable in $\lambda_{Cu}$, i.e., for every type $\tau$, there does not exist a derivation of $\vdash (\lambda x.x \ x) : \tau$.

Exercise: Prove this.

(We’ll see a more systematic way to prove this in Lecture 3).
Church’s type system $\lambda_{Ch}$: annotated terms

- In $\lambda_{Ch}$, types are incorporated explicitly into terms.
- The set of $\mathbb{T}$-annotated terms $\Lambda_\mathbb{T}$ is defined inductively as follows:
  - $x \in \Lambda_\mathbb{T}$ for every term variable $x$.
  - If $M, N \in \Lambda_\mathbb{T}$ then $(M \; N) \in \Lambda_\mathbb{T}$.
  - If $M \in \Lambda_\mathbb{T}$ and $\tau \in \mathbb{T}$ then $(\lambda x : \tau.\; M) \in \Lambda_\mathbb{T}$.
- The notions of type declarations, typing statement, typing contexts and typing judgments are as in Curry’s system.
Reductions in $\lambda_{Ch}$-terms

- Strictly speaking, terms in $\Lambda_T$ are not $\lambda$-terms, so one needs to define its equality theory, reductions, etc.

- The notions of $\beta$-reduction $\rightarrow_{\beta}$, multi-step reduction $\rightarrow_{\beta}^*$ are defined as in untyped $\lambda$-calculus.

  $$(\lambda x : \tau. M)\ N \rightarrow_{\beta} M[x := N].$$

- Church-Rosser property also holds for $T$-annotated terms.
Inference rules of $\lambda_{Ch}$

\[
\Gamma \vdash \lambda_{Ch} \ x : \tau \quad \text{axiom, if } (x : \tau) \in \Gamma
\]

\[
\begin{array}{c}
\Gamma, x : \tau_1 \vdash \lambda_{Ch} M : \tau_2 \\
\hline
\Gamma \vdash \lambda_{Ch} (\lambda x : \tau_1 . M) : (\tau_1 \rightarrow \tau_2)
\end{array}
\quad \rightarrow \text{intro}
\]

Note that $x$ has to be fresh w.r.t. $\Gamma$.

\[
\begin{array}{c}
\Gamma \vdash \lambda_{Ch} M : \tau' \rightarrow \tau \\
\Gamma \vdash \lambda_{Ch} N : \tau'
\hline
\Gamma \vdash \lambda_{Ch} (M \ N) : \tau
\end{array}
\quad \rightarrow \text{elim}
\]
Example

The following are derivable in $\lambda_{Ch}$:

- $\vdash (\lambda x : \tau . x) : (\tau \to \tau)$.
- $\vdash (\lambda x : \sigma \lambda y : \tau . x) : (\sigma \to \tau \to \sigma)$.
- $x : \sigma \vdash (\lambda y : \tau . x) : (\tau \to \sigma)$. 
Theorem

Suppose $\Gamma \vdash_{\lambda_{Ch}} M : \tau$ and $\Gamma \vdash M : \tau'$. Then $\tau \equiv \tau'$.

Proof.

By induction on the structure of $M$.

Exercise: complete the proof.

Type uniqueness does not hold for Curry’s system, but we’ll see that every typeable term in $\lambda_{Cu}$ has a unique principal type.
Relating the Curry and Church systems

- Church’s system can be simulated by Curry’s system by forgetting the type annotations.
- Let $\Lambda$ denote the set of unannotated $\lambda$-terms.
- Define a ‘forgetful map’ $| \cdot |$ from $\Lambda_T$ to $\Lambda$ as follows:

\[
\begin{align*}
|x| & \equiv x \\
|M\ N| & \equiv |M| \ |N| \\
|\lambda x : \tau. M| & \equiv \lambda x.|M|
\end{align*}
\]
Relating the Curry and Church systems

**Theorem (Church to Curry)**

Let $M \in \Lambda_T$. Then $\Gamma \vdash_{\lambda_{Ch}} M : \tau$ implies $\Gamma \vdash_{\lambda_{Cu}} \mid M \mid : \tau$.

Proof: (exercise)

**Theorem (Curry to Church)**

Let $M \in \Lambda$. Then $\Gamma \vdash_{\lambda_{Cu}} M : \tau$ implies that there exists $M' \in \Lambda_T$ such that

$$\Gamma \vdash_{\lambda_{Ch}} M' : \tau \quad \text{and} \quad \mid M' \mid \equiv M.$$

Proof: (exercise)

**Corollary**

A type $\tau \in \mathbb{T}$ is inhabited in $\lambda_{Ch}$ if and only if it is inhabited in $\lambda_{Cu}$.
On-line type checking and inferencing - Haskell

- Haskell: a functional language with types based on simple type theory
- ghci on the departmental systems
- To download it yourself, can use Hugs (adequate and much smaller) https://www.haskell.org/hugs/
- \( \lambda x \lambda y \lambda z. x \, z \, (y \, z) \) is entered as \( \lambda x \rightarrow \lambda y \rightarrow \lambda z \rightarrow x \, z \, (y \, z) \)
- to get the type of this, \( > :t \lambda x \rightarrow \lambda y \rightarrow \lambda z \rightarrow x \, z \, (y \, z) \)
We have covered two styles of type systems for $\lambda$-calculus.

- Church’s system annotates the $\lambda$-terms with types explicitly.
- Curry’s system leaves types implicit.
- Both are essentially the same.
- For next lecture, we shall be studying mainly properties of Curry’s system.