Simply Typed Lambda Calculus

CS3100 Fall 2019

Review

Previously

- Lambda calculus encodings
  - Booleans, Arithmetic, Pairs, Recursion, Lists

Today

- Simply Typed Lambda Calculus

Need for typing

- Consider the untyped lambda calculus
  - false = `λx.λy.y`
  - 0 = `λx.λy.y`
- Since everything is encoded as a function...
  - We can easily misuse terms…
    - false 0 → λy.y
    - if 0 then ...
  - …because everything evaluates to some function
- The same thing happens in assembly language
  - Everything is a machine word (a bunch of bits)
  - All operations take machine words to machine words

How to fix these errors?

Typed Lambda Calculus

- Lambda Calculus + Types → Simply Typed Lambda Calculus ($\lambda^{\rightarrow}$)

Simple Types

$$
\begin{array}{rcll}
A, B & := & \mathrm{B} & \text{(base type)} \\
 & | & A \to B & \text{(functions)} \\
 & | & A \times B & \text{(products)} \\
 & | & 1 & \text{(unit)}
\end{array}
$$

- B is base types like int, bool, float, string, etc.
- $\times$ binds stronger than $\to$
  - $A \times B \to C$ is $(A \times B) \to C$
- $\to$ is right associative.
  - $A \to B \to C$ is $A \to (B \to C)$
  - Same as OCaml
- If we include neither base types nor 1, the system is degenerate. Why?
  - Degenerate = No inhabitant.

Raw Terms

$$
\begin{array}{rcll}
M, N & := & x & \text{(variable)} \\
 & | & M\,N & \text{(application)} \\
 & | & \lambda x : A.\, M & \text{(abstraction)} \\
 & | & \langle M, N \rangle & \text{(pair)} \\
 & | & \mathrm{fst}\, M & \text{(project-1)} \\
 & | & \mathrm{snd}\, M & \text{(project-2)} \\
 & | & (\,) & \text{(unit)}
\end{array}
$$

Typing Judgements

- $M : A$ means that the term M has type A.
- Typing rules are expressed in terms of **typing judgements.**
  - An expression of form $x_1 : A_1, x_2 : A_2, \ldots, x_n : A_n \vdash M : A$
  - Under the assumption $x_1 : A_1, x_2 : A_2, \ldots, x_n : A_n$, M has type A.
  - Assumptions are usually types for free variables in M.
- Use Γ for assumptions.
  - $\Gamma \vdash M : A$
- Assume no repetitions in assumptions.
  - alpha-convert to remove duplicate names.

Quiz

Given $\Gamma, x : A, y : B \vdash M : C$, which of the following is true?

1. $M : C$ holds
2. $x \in \Gamma$
3. $y \notin \Gamma$
4. A and B may be the same type
5. x and y may be the same variable

Quiz

Given $\Gamma, x : A, y : B \vdash M : C$ Which of the following is true?

1. $M : C$ holds ❌ (M may not be a closed term)
2. $x \in \Gamma$ ❌ (Γ has no duplicates)
3. $y \notin \Gamma$ ✅ (Γ has no duplicates)
4. A and B may be the same type ✅ (A and B are type variables)
5. x and y may be the same variable ❌ (Γ has no duplicates)

Typing rules for $\lambda^{\rightarrow}$

$$\frac{}{\Gamma, x : A \vdash x : A}\ (var) \qquad \frac{}{\Gamma \vdash () : 1}\ (unit)$$

$$\frac{\Gamma \vdash M : A \rightarrow B \quad \Gamma \vdash N : A}{\Gamma \vdash M\,N : B}\ (\rightarrow elim) \qquad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x : A.\, M : A \rightarrow B}\ (\rightarrow intro)$$

$$\frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \text{fst } M : A}\ (\times\ elim1) \qquad \frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \text{snd } M : B}\ (\times\ elim2)$$

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B}{\Gamma \vdash \langle M, N \rangle : A \times B}\ (\times\ intro)$$

Typing derivation

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x:A \to A, y:A \vdash x:A \to A}\,(var) \quad \dfrac{\dfrac{}{x:A \to A, y:A \vdash x:A \to A}\,(var) \quad \dfrac{}{x:A \to A, y:A \vdash}}{x:A \to A, y:A \vdash (x\,y):A}}{x:A \to A, y:A \vdash x\,(x\,y):A}}{x:A \to A \vdash (\lambda y:A.\,x\,(x\,y)):A \to A}}{\vdash (\lambda x:A \to A.\,\lambda y:A.\,x\,(x\,y)):(A \to A) \to A \to}$$

Typing derivation

- For each lambda term, there is exactly one type rule that applies.
  - Unique rule to lookup during typing derivation.

Typability

- Not all $\lambda^{\to}$ terms can be assigned a type. For example,
- fst $(\lambda x.\,M)$
- $\langle M, N\rangle\,P$
- Surprisingly, we cannot assign a type for $\lambda x.\,x\,x$ in $\lambda^{\to}$ (or OCaml)
  - x is applied to itself. So its argument type should the type of x!

On fst and snd

In OCaml, we can define `fst` and `snd` as:

In [1]:

```
let fst (a,b) = a
let snd (a,b) = b
```

Out[1]:

```
val fst : 'a * 'b -> 'a = <fun>
```

Out[1]:

```
val snd : 'a * 'b -> 'b = <fun>
```

- Observe that the types are polymorphic.
- But no polymorphism in $\lambda^{\to}$
  - `fst` and `snd` are **keywords** in $\lambda^{\to}$

- For a given type $A \times B$, we can define
  - $(\lambda p : A \times B.\ \text{fst}\ p) : A$
  - $(\lambda p : A \times B.\ \text{snd}\ p) : B$

Reductions in $\lambda^{\rightarrow}$

$$\begin{array}{rrcll}
(\beta_{\rightarrow}) & (\lambda x : A.\, M)\, N & \rightarrow & M[N/x] & \\
(\eta_{\rightarrow}) & \lambda x : A.\, M\, x & \rightarrow & M & \text{if } x \notin FV(M) \\
(\beta_{\times,1}) & \text{fst}\ \langle M, N \rangle & \rightarrow & M & \\
(\beta_{\times,2}) & \text{snd}\ \langle M, N \rangle & \rightarrow & N & \\
(\eta_{\times}) & \langle \text{fst}\ M, \text{snd}\ M \rangle & \rightarrow & M &
\end{array}$$

$$(cong1)\ \frac{M \rightarrow M'}{M\, N \rightarrow M'\, N} \qquad (cong2)\ \frac{N \rightarrow N'}{M\, N \rightarrow M\, N'}$$

$$(\xi)\ \frac{M \rightarrow M'}{\lambda x : A.\, M \rightarrow \lambda x : A.\, M'}$$

Type Soundness

- Well-typed programs do not get **stuck**.
  - stuck = not a value & no reduction rule applies.
  - $\text{fst}\ (\lambda x.\, x)$ is stuck.
  - ( ) ( ) is stuck.
- In practice, well-typed programs do not have runtime errors.

Theorem (Type Soundness). If $\vdash M : A$ and $M \rightarrow M'$, then either M' is a value or there exists an M'' such that $M' \rightarrow M''$.

Proved using two lemmas **progress** and **preservation**.

Preservation

If a term M is well-typed, and M can take a step to M' then M is well-typed.

Lemma (Preservation). If $\vdash M : A$ and $M \rightarrow M'$, then $\vdash M' : A$.

Proof is by induction on the reduction relation $M \rightarrow M'$.

Preservation : Case $\beta_{\rightarrow}$

Lemma (Preservation). If $\vdash M : A$ and $M \rightarrow M'$, then $\vdash M' : A$.

Recall, $(\beta_{\rightarrow})$ rule is $(\lambda x : A. M_1)\ N \rightarrow M_1[N/x]$.

Assume $\vdash M : A$. Here $M = (\lambda x : B. M_1)\ N$ and $M' = M_1[N/x]$.

We know M is well-typed. And from the typing derivation know that $x : B \vdash M_1 : A$ and $\vdash N : B$.

Lemma (substitution). If $x : B \vdash M : A$ and $\vdash N : B$, then $\vdash M[N/x] : A$.

By substitution lemma, $\vdash M_1[N/x] : A$. Therefore, preservation holds.

Progress

Progress says that if a term M is well-typed, then either M is a value, or there is an M' such that M can take a step to M'.

Lemma (Progress). If $\vdash M : A$ then either M is a value or there exists an M' such that $M \rightarrow M'$.

Proof is by induction on the derivation of $\vdash M : A$.

- Case var does not apply
- Cases $unit$, $\times\ intro$ and $\rightarrow intro$ are trivial; they are values.
- Reduction is possible in other cases as M is well-typed.

Type Safety = Progress + Preservation

Expressive power of $\lambda^{\rightarrow}$

- Clearly, not all untyped lambda terms are well-typed.
  - Any term that gets stuck is ill-typed.
- Are there terms that are ill-typed but do not get stuck?

- Unfortunately, the answer is yes!
  - Consider $\lambda x. x$. In $\lambda^{\rightarrow}$, we must assign type for x
  - Pick a concrete type for x

- $\lambda x : 1 . x$.
  - $(\lambda x : 1 . x)\ \langle (\,), (\,) \rangle$ is ill-typed, but does not get stuck.

Expressive power of $\lambda^{\rightarrow}$

- As mentioned earlier, we can no longer write recursive function.
  - $(\lambda x.\, x\, x)\ (\lambda x.\, x\, x)$
- Every well-typed $\lambda^{\rightarrow}$ term terminates!
  - $\lambda^{\rightarrow}$ is strongly normalising.

Connections to propositional logic

Consider the following types

$$\begin{array}{ll}
(1) & (A \times B) \rightarrow A \\
(2) & A \rightarrow B \rightarrow (A \times B) \\
(3) & (A \rightarrow B) \rightarrow (B \rightarrow C) \rightarrow (A \rightarrow C) \\
(4) & A \rightarrow A \rightarrow A \\
(5) & ((A \rightarrow A) \rightarrow B) \rightarrow B \\
(6) & A \rightarrow (A \times B) \\
(7) & (A \rightarrow C) \rightarrow C
\end{array}$$

Can you find closed terms of these types?

Connections to propositional logic

$$\begin{array}{lll}
(1) & (A \times B) \rightarrow A & \lambda x : A \times B.\ \text{fst}\ x \\
(2) & A \rightarrow B \rightarrow (A \times B) & \lambda x : A.\ \lambda y : B.\ \langle x, y \rangle \\
(3) & (A \rightarrow B) \rightarrow (B \rightarrow C) \rightarrow (A \rightarrow C) & \lambda x : A \rightarrow B.\ \lambda y : B \rightarrow C.\ \lambda z : A.\ y\ (x\ z) \\
(4) & A \rightarrow A \rightarrow A & \lambda x : A.\ \lambda y : A.\ x \\
(5) & ((A \rightarrow A) \rightarrow B) \rightarrow B & \lambda x : (A \rightarrow A) \rightarrow B.\ x\ (\lambda y : A.\ y) \\
(6) & A \rightarrow (A \times B) & \text{can't find a closed term} \\
(7) & (A \rightarrow C) \rightarrow C & \text{can't find a closed term}
\end{array}$$

A different question

- Given a type, whether there exists a closed term for it?
- Replace $\rightarrow$ with $\implies$ and $\times$ with $\wedge$.

$$
\begin{array}{ll}
(1) & (A \wedge B) \implies A \\
(2) & A \implies B \implies (A \wedge B) \\
(3) & (A \implies B) \implies (B \implies C) \implies (A \implies C) \\
(4) & A \implies A \implies A \\
(5) & ((A \implies A) \implies B) \implies B \\
(6) & A \implies (A \wedge B) \\
(7) & (A \implies C) \implies C
\end{array}
$$

What can we say about the validity of these logical formulae?

A different question

$$
\begin{array}{ll}
(1) & (A \wedge B) \implies A \\
(2) & A \implies B \implies (A \wedge B) \\
(3) & (A \implies B) \implies (B \implies C) \implies (A \implies C) \\
(4) & A \implies A \implies A \\
(5) & ((A \implies A) \implies B) \implies B \\
(6) & A \implies (A \wedge B) \\
(7) & (A \implies C) \implies C
\end{array}
$$

$(1) - (5)$ are valid, (6) and (7) are not!

Proving a propositional logic formula

- How to prove $(A \wedge B) \implies A$?
  - Assume $A \wedge B$ holds. By the first conjunct, A holds. Hence, the proof.
- Consider the program $\lambda x : A \times B.\ \mathrm{fst}\ x$.
  - Observe the close correspondence of this **program** to the **proof**.
- What is the type of this program? $(A \times B) \rightarrow A$.
  - Observe the close correspondence of this **type** to the **proposition**.
- Curry-Howard correspondence between $\lambda^{\rightarrow}$ and propositional logic.

Curry-Howard Correspondence

- Proposition:Proof :: Type:Program
- Intuitionistic/constructive logic and not classical logic
  - Law of excluded middle ($A \vee \neg A$) does not hold for an arbitrary A.
    - Can't prove by contradiction
  - In order to prove, *construct* the proof object!

Propositional Intuitionistic Logic

$$\text{Formulas:} A, B ::= \alpha \mid A \rightarrow B \mid A \wedge B \mid \top$$

where α is atomic formulae.

A derivation is

$$x_1 : A_1, x_2 : A_2, \ldots, x_n : A_n \vdash A$$

where $A_1, A_2, \ldots$ are assumptions, $x_1, x_2, \ldots$ are names for those assumptions and A is the formula derived from those assumptions.

Derivations through natural deduction

$$\frac{}{\Gamma, x : A \vdash x : A}\ (axiom) \qquad \frac{}{\Gamma \vdash \top}\ (\top\ intro)$$

$$\frac{\Gamma \vdash A \implies B \quad \Gamma \vdash A}{\Gamma \vdash B}\ (\implies elim) \qquad \frac{\Gamma, x : A \vdash B}{\Gamma \vdash A \implies B}\ (\implies intro)$$

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\ (\wedge\ elim1) \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\ (\wedge\ elim2)$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ (\wedge\ intro)$$

Curry Howard Isomorphism

- Allows one to switch between type-theoretic and proof-theoretic views of the world at will.
  - used by theorem provers and proof assistants such as coq, HOL/Isabelle, etc.
- Reductions of $\lambda^{\rightarrow}$ terms corresponds to proof simplification.

Curry Howard Isomorphism

$\lambda^{\rightarrow}$	Propositional Intuitionistic Logic
Types	Propositions
1	$\top$
$\times$	$\wedge$
$\rightarrow$	$\implies$
Programs	Proofs
Reduction	Proof Simplification

What about $\vee$?

Disjunction

Extend the logic with:

$$\text{Formulas: } A, B ::= \ldots \mid A \vee B \mid \bot$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\ (\vee\ intro1) \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\ (\vee\ intro2)$$

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash C}\ (\bot\ elim) \qquad \frac{\Gamma \vdash A \vee B \quad \Gamma, x{:}A \vdash C \quad \Gamma, y{:}B \vdash C}{\Gamma \vdash C}\ (\vee\ elim)$$

Sum Types

Extend $\stlc$ with:

\[\begin{array}{rrcl} \text{Simple Types: } & A,B & ::= & \ldots \mid A + B \mid 0 \\ \text{Raw Terms: } & M,N,P & ::= & \ldots \mid \case{M}{x:A}{N}{y:B}{P} \\ & & \mid & \inl{B}{M} \mid \inr{A}{M} \mid \square_{A} ~M \end{array} \]

The OCaml equivalent of this sum type is:

```
type ('a,'b) either = Inl of 'a | Inr of 'b
```

- Similar to `fst` and `snd`, there is no polymorphism in $\stlc$.
  - Hence, `inl` and `inr` are keywords.

Explicit Type Annotation for `inl` and `inr`

$$\begin{array}{rrcl} \text{Raw Terms:} & M, N, P & ::= & \ldots \mid \text{case } M \text{ of inl } x{:}A \Rightarrow N \mid \text{inl } y{:}B \Rightarrow P \\ & & \mid & \text{inl } [B]\ M \mid \text{inr } [A]\ M \mid \square_A M \end{array}$$

- Observe that the term for inl and inr require explicit type annotation.
- Without that $\text{inl}\ ()$ has many possible types captured by $1 + A$.
  - Bottom up type checking is not possible as A is left undefined.
    - No type inference or polymorphism in $\lambda^{\rightarrow}$.
- Add explicit annotation and preserve bottom-up type checking property.

Sum Types : Contradiction

Extend $\stlc$ with:

\[\begin{array}{rrcl} \text{Simple Types: } & A,B & ::= & \ldots \mid A + B \mid 0 \\ \text{Raw Terms: } & M,N,P & ::= & \ldots \mid \case{M}{x:A}{N}{y:B}{P} \\ & & \mid & \inl{B}{M} \mid \inr{A}{M} \mid \square_{A} ~M \end{array} \]

- The type 0 is an **uninhabited** type
  - There are no values of this type.
- The OCaml equivalent is an empty variant type:

```
type zero = |
```

Sum Types : Static Semantics

Extend $\lambda^{\rightarrow}$ with:

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \text{inl } [B]\ M : A + B}\ (+\ intro1) \qquad \frac{\Gamma \vdash M : B}{\Gamma \vdash \text{inr } [A]\ M : A + B}\ (+\ intro2)$$

$$\frac{\Gamma \vdash M : A + B \quad \Gamma, x : A \vdash N : C \quad \Gamma, y : B \vdash P : C}{\Gamma \vdash \text{case } M \text{ of inl } x : A \Rightarrow N \mid \text{inl } y : B \Rightarrow P : C}\ (+\ elim)$$

$$\frac{\Gamma \vdash M : 0}{\Gamma \vdash \square_A M : A}\ (\square)$$

Casting and type soundness

- Recall, Type Soundness = Progress + Preservation
- But $\square_A$ changes the type of the expression
  - Is type soundness lost?
- Consider $\lambda x : 0.(\square_{1 \rightarrow 1}\ x)\ ()$
  - This term is well-typed.
  - If we are able to call this function, the program would crash since x is not a function.
- There is no way to call this function since the type 0 is uninhabited!
  - Type Soundness is preserved.

Sum Types : Dynamic Semantics

Extend $\rightarrow$ with:

$$\frac{M \to M'}{\text{case } M \text{ of inl } x_1 : A \Rightarrow N_1 \mid \text{inl } x_2 : B \Rightarrow N_2 \to \text{case } M' \text{ of inl } x_1 : A \Rightarrow N_1 \mid \text{inl } x_2 : B \Rightarrow N_2}$$

$$\frac{M = \text{inl } [B]\, M'}{\text{case } M \text{ of inl } x_1 : A \Rightarrow N_1 \mid \text{inl } x_2 : B \Rightarrow N_2 \to N_1[M'/x_1]}$$

$$\frac{M = \text{inr } [A]\, M'}{\text{case } M \text{ of inl } x_1 : A \Rightarrow N_1 \mid \text{inl } x_2 : B \Rightarrow N_2 \to N_2[M'/x_2]}$$

Type Erasure

- Although we carry around type annotations during reduction, we do not examine them.
  - No runtime type checking to see if function is applied to appropriate arguments, etc.
- Most compilers drop the types in the compiled form of the program (**erasure**).

$$\begin{array}{rcl} \text{erase}(x) & = & x \\ \text{erase}(M\, N) & = & \text{erase}(M)\ \text{erase}(N) \\ \text{erase}(\lambda x : A.\, M) & = & \lambda x.\ \text{erase}(M) \\ \text{erase}(\text{inr } [A]\, M) & = & \text{erase}(\text{inr erase}(M)) \end{array}$$

etc.

Type erasure

Theorem (Type erasure).

1. If $M \to M'$ under the $\lambda^{\to}$ reduction relation, then $\text{erase}(M) \to \text{erase}(M')$ under untyped reduction relation.
2. If $\text{erase}(M) \to N'$ under the untyped reduction relation, then there exists a $\lambda^{\to}$ term M' such that $M \to M'$ under $\lambda^{\to}$ reduction relation and $\text{erase}(M') = N'$.

Static vs Dynamic Typing

- OCaml, Haskell, Standard ML are **statically typed languages**.
  - Only well-typed programs are allowed to run.
  - Type soundness holds; well-typed programs do no get stuck.
  - Types can be erased at compilation time.

- What about Python, JavaScript, Clojure, Perl, Lisp, R, etc?
  - **Dynamically typed languages.**
  - No type checking at compile time; anything goes.
    - `x = lambda a : a + 10; x("Hello")` is a runtime error.
  - Allows more programs to run, but types need to be checked at runtime.
    - Types cannot be erased!

Fin