CS720

Logical Foundations of Computer Science

Lecture 18: Simply Typed Lambda Calculus (STLC)

Tiago Cogumbreiro

What we have learned so far

1. Learned a new programming language called Coq
2. Exercised logic programming and functional programming
3. Formal proof development (inductive definitions, proofs by induction, case analysis)
4. Specify mathematically software systems (ie, formalization process)
5. Designed an imperative programming language (control flow, memory)
6. Proved the correctness of compiler transformations (constant folding)
7. Designed a program verification system (Hoare logic)
8. Proved the correctness of programs (using Hoare logic)
9. Formalized a static analysis (type system)

Overview: λ-calculus

- We want to module the key concept of ***functional abstraction***
- We want to represent: function calls, procedures, methods
- We will follow the same pattern:
  1. Language
  2. Small-step semantics
  3. Typing rules
  4. Show Progress and Type Preservation
- Introduce: ***variable binding*** and ***substitution***

Syntax

```
t ::= x                          variable
    | \x:T1.t2                   abstraction
    | t1 t2                      application
    | true                       constant true
    | false                      constant false
    | if t1 then t2 else t3      conditional

T ::= Bool
    | T1 → T2
```

- Our language accepts ***higher-order*** functions
- All functions are anonymous.
- Only booleans (to simplify presentation)

Examples

`\x:Bool. x`

Examples

`\x:Bool. x`

> The identity function for booleans.

`\x:Bool. if x then false else true`

Examples

```
\x:Bool. x
```

> The identity function for booleans.

```
\x:Bool. if x then false else true
```

> The boolean "not" function.

```
(\f:Bool→Bool. f (f true)) (\x:Bool. false)
```

Examples

`\x:Bool. x`

> The identity function for booleans.

`\x:Bool. if x then false else true`

> The boolean "not" function.

`(\f:Bool→Bool. f (f true)) (\x:Bool. false)`

> The same higher-order function, applied to the constantly `false` function.

What is the type of?

```
\x:Bool. x
```

What is the type of?

```
\x:Bool. x
```

> `Bool→Bool`

```
\x:Bool. if x then false else true
```

What is the type of?

`\x:Bool. x`

> `Bool→Bool`

`\x:Bool. if x then false else true`

> `Bool→Bool`

`(\f:Bool→Bool. f (f true)) (\x:Bool. false)`

What is the type of?

`\x:Bool. x`

> `Bool→Bool`

`\x:Bool. if x then false else true`

> `Bool→Bool`

`(\f:Bool→Bool. f (f true)) (\x:Bool. false)`

> `Bool`

Which are values?

A variable: `x`

Which are values?

A variable: `x`

> No.

Function definition: `\x:T1.t2`

Which are values?

A variable: `x`

> No.

Function definition: `\x:T1.t2`

> **Yes!**

Function application: `t1 t2`

Which are values?

A variable: `x`

> No.

Function definition: `\x:T1.t2`

> **Yes!**

Function application: `t1 t2`

> No.

Booleans: `true`, `false`

Which are values?

A variable: `x`

> No.

Function definition: `\x:T1.t2`

> **Yes!**

Function application: `t1 t2`

> No.

Booleans: `true`, `false`

> Yes.

Conditional: `if t1 then t2 else t3`

Which are values?

A variable: `x`

> No.

Function definition: `\x:T1.t2`

> **Yes!**

Function application: `t1 t2`

> No.

Booleans: `true`, `false`

> Yes.

Conditional: `if t1 then t2 else t3`

> No.

Values

$$\frac{}{\text{value}(\lambda x\colon T.t)}\,(\text{v-abs}) \qquad \frac{}{\text{value}(\texttt{true})}\,(\text{v-true}) \qquad \frac{}{\text{value}(\texttt{false})}\,(\text{v-false})$$

Values

$$\frac{}{\mathrm{value}(\lambda x\colon T.t)}\,(\text{v-abs}) \qquad \frac{}{\mathrm{value}(\mathtt{true})}\,(\text{v-true}) \qquad \frac{}{\mathrm{value}(\mathtt{false})}\,(\text{v-false})$$

Or, formally:

```
Inductive value : tm → Prop :=
  | v_abs : forall x T t,
      value (tabs x T t)
  | v_true :
      value ttrue
  | v_false :
      value tfalse.
```

Variable substitution

We need to represent variable substitution, because of function application.

```
(\ x:Bool. if x then true else x) false
```

Becomes:

```
if false then true else false
```

Variable substitution

> We need to represent variable substitution, because of function application.

```
(\ x:Bool. if x then true else x) false
```

Becomes:

```
if false then true else false
```

> Let us define a substitution operator `[x:=v]` such that when applied to a term `t`, it replaces any occurrence of `x` in `t` by `v`. Care must be taken when handling function abstractions.

In the above example, we would have

```
[x := false] (if x then true else x) = if false then true else false
```

Substituition examples

```
[x:=true] x
```

```
true
```

Substituition examples

```
[x:=true] x
```

```
true
```

```
[x:=true] (if x then x else y)
```

Substituition examples

```
[x:=true] x
```

```
true
```

```
[x:=true] (if x then x else y)
```

```
if true then true else y
```

```
[x:=true] y
```

Substituition examples

```
[x:=true] x
```

```
true
```

```
[x:=true] (if x then x else y)
```

```
if true then true else y
```

```
[x:=true] y
```

```
y
```

Substituition examples

```
[x:=true] false
```

Substituition examples

```
[x:=true] false
```

```
false
```

```
[x:=true] (\y:Bool. if y then x else false)
```

Substituition examples

```
[x:=true] false
```

false

```
[x:=true] (\y:Bool. if y then x else false)
```

\y:Bool. if y then true else false

```
[x:=true] (\x:Bool. x)
```

Substituition examples

```
[x:=true] false
```

```
false
```

```
[x:=true] (\y:Bool. if y then x else false)
```

```
\y:Bool. if y then true else false
```

```
[x:=true] (\x:Bool. x)
```

```
\x:Bool. x
```

Let us implement subst

Formal unit tests:

```
Fixpoint subst (x:string) (s:tm) (t:tm) : tm :=
(* TODO *)

Goal subst x ttrue (tvar x) = ttrue. auto. Qed.
Goal subst x ttrue (tif (tvar x) (tvar x) (tvar y)) = tif ttrue ttrue (tvar y). auto. Qed.
Goal subst x ttrue (tvar y) = tvar y. auto. Qed.
Goal subst x ttrue tfalse = tfalse. auto. Qed.
Goal subst x ttrue (tabs y TBool (tif (tvar y) (tvar x) tfalse)) = tabs y TBool (tif (tvar
Goal subst x ttrue (tabs y TBool (tvar x)) = (tabs y TBool ttrue). auto. Qed.
Goal subst x ttrue (tabs y TBool (tvar y)) = (tabs y TBool (tvar y)). auto. Qed.
Goal subst x ttrue (tabs x TBool (tvar x)) = (tabs x TBool (tvar x)). auto. Qed.
```

Variable substitution limitation

What does this produce?

```
[ x := y ] (\y:Bool. x)
```

Variable substitution limitation

What does this produce?

```
[ x := y ] (\y:Bool. x)
(\y:Bool. y)
```

> We consider such a substitution to be ill-formed. That is because `y` is a "global" variable defined outside the scope of function `\y. Bool. x`. After substitution, however, variable `y` refers to the parameter `y` of `\y. Bool. x` which can cause subtle bugs.
> In our language, we only care about programs where all variables are introduced via some function abstraction, which negates this situation. Such programs are known as ***closed*** programs, and these "global" variables, which are not introduced by function abstractions are known as ***free*** variables.

UMass Boston

Small-step semantics

$$\frac{}{\texttt{if true then } t_1 \texttt{ else } t_2 \Rightarrow t_1}\text{(if-true)} \qquad \frac{}{\texttt{if false then } t_1 \texttt{ else } t_2 \Rightarrow t_2}\text{(if-false)}$$

$$\frac{t_1 \Rightarrow t_1'}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \Rightarrow \texttt{if } t_1' \texttt{ then } t_2 \texttt{ else } t_3}\text{(if)}$$

Small-step semantics

$$\frac{}{\texttt{if true then } t_1 \texttt{ else } t_2 \Rightarrow t_1}\text{(if-true)} \qquad \frac{}{\texttt{if false then } t_1 \texttt{ else } t_2 \Rightarrow t_2}\text{(if-false)}$$

$$\frac{t_1 \Rightarrow t_1'}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \Rightarrow \texttt{if } t_1' \texttt{ then } t_2 \texttt{ else } t_3}\text{(if)}$$

$$\frac{\text{value}(v_2)}{(\lambda x : T.t_1)v_2 \Rightarrow [x := v_2]t_1}\text{(app-abs)}$$

$$\frac{t_1 \Rightarrow t_1'}{t_1\ t_2 \Rightarrow t_1'\ t_2}\text{(app1)} \qquad \frac{\text{value}(v_1) \qquad t_2 \Rightarrow t_2'}{v_1\ t_2 \Rightarrow v_1\ t_2'}\text{(app2)}$$

Exercise

- What is the type of this expression?

```
\x: Bool. y
```

Exercise

> What is the type of this expression?

```
\x: Bool. y
```

- It depends on the type of `y`.
- What is the type of `y`?
- How do we typecheck such an expression?

Type system

$$\frac{}{\Gamma \vdash \texttt{true} \in \texttt{Bool}}\,(\text{T-true}) \qquad \frac{}{\Gamma \vdash \texttt{false} \in \texttt{Bool}}\,(\text{T-false})$$

$$\frac{\Gamma \vdash t_1 \in \texttt{Bool} \qquad \Gamma \vdash t_2 \in T \qquad \Gamma \vdash t_3 \in T}{\Gamma \vdash \texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 \in T}\,(\text{T-if})$$

Type system

$$\frac{}{\Gamma \vdash \texttt{true} \in \texttt{Bool}}\text{(T-true)} \qquad \frac{}{\Gamma \vdash \texttt{false} \in \texttt{Bool}}\text{(T-false)}$$

$$\frac{\Gamma \vdash t_1 \in \texttt{Bool} \qquad \Gamma \vdash t_2 \in T \qquad \Gamma \vdash t_3 \in T}{\Gamma \vdash \texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 \in T}\text{(T-if)}$$

$$\frac{\Gamma(x) = T}{\Gamma \vdash x \in T}\text{(T-var)} \qquad \frac{\Gamma \& \{x \mapsto T_p\} \vdash t_b \in T_b}{\Gamma \vdash \lambda x\colon T_p.t_b \in T_p \to T_b}\text{(T-abs)}$$

$$\frac{\Gamma \vdash t_f \in T_a \to T_r \qquad \Gamma \vdash t_a \in T_a}{\Gamma \vdash t_f\ t_a \in T_r}\text{(T-app)}$$

Summary

- Introduced the Simply-Typed λ-Calculus
- Introduced ***variable binding*** and ***substitution***
- Formalized function (declaration and application) semantics