软件科学基础

STLC: The Simply Typed Lambda-Calculus

熊英飞
北京大学

复习：λ演算

- 用函数调用定义计算
- 是现代（函数式）程序设计语言的理论基础
- 现代程序设计语言的语法和语义通常在λ演算的基础上扩充而成

命令式语言	函数式语言
计算由命令的执行构成	计算由函数调用构成
函数（过程）只是命令的包装	命令只是函数调用的特殊形式
代码和数据分离	代码和数据统一

复习：语法

t	::=		*terms:*
		x	*variable*
		λx.t	*abstraction*
		t t	*application*

复习：语义

- Alpha-Renaming：绑定的变量可以随意改名
  - 如：$(\lambda x.x)\,(\lambda x.x) = (\lambda y.y)\,(\lambda z.z)$
- Beta-Reduction：即函数调用，也是唯一的计算步骤

$$(\lambda x.\,t_{12})\;t_2 \longrightarrow [x \mapsto t_2]t_{12},$$

  - 如： $(\lambda y.y)\,(\lambda z.z) \to (\lambda z.z)$

扩展λ演算描述编程语言

- 在λ演算的基础上扩展程序设计语言其他部分
  - 理论上也可以将语言转换到λ演算，但过于间接
- 本讲介绍在多篇教材中使用的基于λ演算扩展的简单函数式编程语言STLC
  - Simply Typed Lambda Calculus
- STLC vs IMP
  - 函数式、带类型
- 历史
  - Church在1940年发表论文“A Formulation Of The Simple Theory Of Types”，构建带类型的λ演算
  - Peter Landin在1965年的论文 “A Correspondence between ALGOL 60 and Church’s Lambda-notation”，建立语言设计和λ演算之间的联系

Lambda立方体

```
                          Calculus of Constructions
 type operators +--------+
               /|       /|
              / |      / |
polymorphism +--------+  |
             |  |     |  |
             |  +-----|--+
             | /      | /
             |/       |/
             +--------+ dependent types
           STLC
```

语法和类型

```
t ::= x (variable)
    | \x:T,t (abstraction)
    | t t (application)
    | true (constant true)
    | false (constant false)
    | if t then t else t (conditional)
```

采用逗号以便和Coq兼容

```
T ::= Bool
    | T → T
```

Coq定义

```
Inductive ty : Type :=
  | Ty_Bool  : ty
  | Ty_Arrow : ty -> ty -> ty.
```

```
Inductive tm : Type :=
  | tm_var   : string -> tm
  | tm_app   : tm -> tm -> tm
  | tm_abs   : string -> ty -> tm -> tm
  | tm_true  : tm
  | tm_false : tm
  | tm_if    : tm -> tm -> tm -> tm.
```

语法解析-类型

```
Declare Custom Entry stlc_ty.
Notation "x" := x (in custom stlc_ty at level 0, x global) :
stlc_scope.

Notation "<{{ x }}>" := x (x custom stlc_ty).

Notation "( t )" := t (in custom stlc_ty at level 0, t custom
stlc_ty) : stlc_scope.
Notation "S -> T" := (Ty_Arrow S T) (in custom stlc_ty at level 99,
right associativity) : stlc_scope.

Notation "$( t )" := t (in custom stlc_ty at level 0, t constr) :
stlc_scope.
……
```

语法解析-类型

```
Notation "'Bool'" := Ty_Bool (in custom stlc_ty at level 0) :
stlc_scope.
Notation "'if' x 'then' y 'else' z" :=
  (tm_if x y z) (in custom stlc_tm at level 200,
                    x custom stlc_tm,
                    y custom stlc_tm,
                    z custom stlc_tm at level 200,
                    left associativity).
Notation "'true'"  := true (at level 1).
Notation "'true'"  := tm_true (in custom stlc_tm at level 0).
Notation "'false'"  := false (at level 1).
Notation "'false'"  := tm_false (in custom stlc_tm at level 0).
```

语法解析-表达式

```
Notation "$( x )" := x (in custom stlc_tm at level 0, x constr,
only parsing) : stlc_scope.
Notation "x" := x (in custom stlc_tm at level 0, x constr at
level 0) : stlc_scope.
Notation "<{ e }>" := e (e custom stlc_tm at level 200) :
stlc_scope.
Notation "( x )" := x (in custom stlc_tm at level 0, x custom
stlc_tm) : stlc_scope.

Notation "x y" := (tm_app x y) (in custom stlc_tm at level 10,
left associativity) : stlc_scope.
Notation "\ x : t , y" :=
  (tm_abs x t y) (in custom stlc_tm at level 200, x global,
                     t custom stlc_ty,
                     y custom stlc_tm at level 200,
                     left associativity).
```

语法解析

```
Definition x : string := "x".
Definition y : string := "y".
Definition z : string := "z".
Hint Unfold x : core.
Hint Unfold y : core.
Hint Unfold z : core.
```

小步法操作语义

- 值:
  - 定义正常计算结束的结果
  - \x: Bool, if true then x else false是值吗?

- 可以不是，如在Coq中
  - `Compute (fun x:bool => if true then x else false).`
  - `(* = fun x : bool => x : bool -> bool*)`
- 但通常是。其他多数语言不会在没传参的时候就开始计算一个函数定义
  - 同时，定义为值可以简化后续定义，避免考虑函数调用时的alpha-renaming问题（稍后解释）
- STLC将任意lambda抽象定义为值

值

```
Inductive value : tm -> Prop :=
  | v_abs : forall x T2 t1,
      value <{\x:T2, t1}>
  | v_true :
      value <{true}>
  | v_false :
      value <{false}>.

Hint Constructors value : core.
```

代换

- 在beta-reduction的时候需要将形参代换为实参

```
Fixpoint subst (x : string) (s : tm) (t : tm) : tm :=
  match t with
  | tm_var y =>
      if String.eqb x y then s else t
  | <{\y:T, t1}> =>
      if String.eqb x y then t else <{\y:T, [x:=s] t1}>
  | <{t1 t2}> =>
      <{[x:=s] t1 [x:=s] t2}>
  | <{true}> => <{true}>
  | <{false}> => <{false}>
  | <{if t1 then t2 else t3}> =>
      <{if [x:=s] t1 then [x:=s] t2 else [x:=s] t3}>
  end

where "'[' x ':=' s ']' t" := (subst x s t) (in custom stlc).
```

小步法操作语义

$$\frac{\mathtt{value}\ v_2}{(\backslash x{:}T_2, t_1)\ v_2 \rightarrow [x{:}=v_2]t_1}\quad \text{(ST\_AppAbs)}$$

$$\frac{t_1 \rightarrow t_1'}{t_1\ t_2 \rightarrow t_1'\ t_2}\quad \text{(ST\_App1)}$$

$$\frac{\begin{array}{c}\mathtt{value}\ v_1 \\ t_2 \rightarrow t_2'\end{array}}{v_1\ t_2 \rightarrow v_1\ t_2'}\quad \text{(ST\_App2)}$$

$$\frac{}{(\mathtt{if}\ \mathtt{true}\ \mathtt{then}\ t_1\ \mathtt{else}\ t_2) \rightarrow t_1}\quad \text{(ST\_IfTrue)}$$

$$\frac{}{(\mathtt{if}\ \mathtt{false}\ \mathtt{then}\ t_1\ \mathtt{else}\ t_2) \rightarrow t_2}\quad \text{(ST\_IfFalse)}$$

$$\frac{t_1 \rightarrow t_1'}{(\mathtt{if}\ t_1\ \mathtt{then}\ t_2\ \mathtt{else}\ t_3) \rightarrow (\mathtt{if}\ t_1'\ \mathtt{then}\ t_2\ \mathtt{else}\ t_3)}\quad \text{(ST\_If)}$$

Coq定义

```
Inductive step : tm -> tm -> Prop :=
  | ST_AppAbs : forall x T2 t1 v2,
         value v2 ->
         <{(\x:T2, t1) v2}> --> <{ [x:=v2]t1 }>
  | ST_App1 : forall t1 t1' t2,
         t1 --> t1' ->
         <{t1 t2}> --> <{t1' t2}>
  | ST_App2 : forall v1 t2 t2',
         value v1 ->
         t2 --> t2' ->
         <{v1 t2}> --> <{v1  t2'}>
```

Coq定义

```
  | ST_IfTrue : forall t1 t2,
      <{if true then t1 else t2}> --> t1
  | ST_IfFalse : forall t1 t2,
      <{if false then t1 else t2}> --> t2
  | ST_If : forall t1 t1' t2 t3,
      t1 --> t1' ->
      <{if t1 then t2 else t3}> --> <{if t1' then t2 else t3}>

where "t '-->' t'" := (step t t').

Hint Constructors step : core.

Notation multistep := (multi step).
Notation "t1 '-->*' t2" := (multistep t1 t2) (at level 40).
```

考虑允许约简函数定义

如果修改value定义，令value和标准型等价，
并添加如下运算规则后，会出现什么问题?

$$\frac{t_1 \to t_2}{\backslash x{:}T, t_1 \to \backslash x{:}T, t_2}$$

$$\frac{\text{value } v_2}{(\backslash x{:}T_2, t_1)\ v_2 \to [x{:=}v_2]t_1} \quad \text{(ST\_AppAbs)}$$

$$\frac{t_1 \to t_1'}{t_1\ t_2 \to t_1'\ t_2} \quad \text{(ST\_App1)}$$

$$\frac{\text{value } v_1 \qquad t_2 \to t_2'}{v_1\ t_2 \to v_1\ t_2'} \quad \text{(ST\_App2)}$$

出错的情况

- \y:Bool, ((\x:Bool, (\y:Bool, x)) y)
- → \y:Bool, (\y:Bool, y)

- 正确答案： \y:Bool, (\z:Bool, y)
- 解决该问题需要引入alpha-renaming，本课程不涉及

类型系统

- 必须知道变量的类型才能对带变量的表达式进行类型检查
- 引入上下文Gamma，即从变量名到类型的映射
  - 因此，类型正确的程序不是上下文无关语言
  - 需要引入额外规则来保证类型正确性
- 引入三元类型推导关系
  - Gamma ⊢ t ∈ T
  - 在Gamma下，t具有T类型

类型推导规则

$$\frac{\text{Gamma } x = T_1}{\text{Gamma} \vdash x \in T_1} \quad \text{(T\_Var)}$$

$$\frac{x \mapsto T_2 \ ; \ \text{Gamma} \vdash t_1 \in T_1}{\text{Gamma} \vdash \backslash x{:}T_2, t_1 \in T_2\text{->}T_1} \quad \text{(T\_Abs)}$$

$$\frac{\begin{array}{c}\text{Gamma} \vdash t_1 \in T_2\text{->}T_1 \\ \text{Gamma} \vdash t_2 \in T_2\end{array}}{\text{Gamma} \vdash t_1\ t_2 \in T_1} \quad \text{(T\_App)}$$

$$\frac{}{\text{Gamma} \vdash \text{true} \in \text{Bool}} \quad \text{(T\_True)}$$

$$\frac{}{\text{Gamma} \vdash \text{false} \in \text{Bool}} \quad \text{(T\_False)}$$

$$\frac{\text{Gamma} \vdash t_1 \in \text{Bool} \qquad \text{Gamma} \vdash t_2 \in T_1 \qquad \text{Gamma} \vdash t_3 \in T_1}{\text{Gamma} \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 \in T_1} \quad \text{(T\_If)}$$

Coq定义

```
Inductive has_type : context -> tm -> ty -> Prop :=
  | T_Var : forall Gamma x T1,
      Gamma x = Some T1 ->
      <{ Gamma |-- x \in T1 }>
  | T_Abs : forall Gamma x T1 T2 t1,
      <{ x |-> T2 ; Gamma |-- t1 \in T1 }> ->
      <{ Gamma |-- \x:T2, t1 \in T2 -> T1 }>
  | T_App : forall T1 T2 Gamma t1 t2,
      <{ Gamma |-- t1 \in T2 -> T1 }> ->
      <{ Gamma |-- t2 \in T2 }> ->
      <{ Gamma |-- t1 t2 \in T1 }>
```

Coq定义

```
| T_True : forall Gamma,
    <{ Gamma |-- true \in Bool }>
| T_False : forall Gamma,
    <{ Gamma |-- false \in Bool }>
| T_If : forall t1 t2 t3 T1 Gamma,
     <{ Gamma |-- t1 \in Bool }> ->
     <{ Gamma |-- t2 \in T1 }> ->
     <{ Gamma |-- t3 \in T1 }> ->
     <{ Gamma |-- if t1 then t2 else t3 \in T1 }>

where "Gamma '|--
' t '\in' T" := (has_type Gamma t T).
```

作业

- 完成STLC中standard非optional的3道习题以及typing_nonexample_3
  - 请使用最新英文版教材