



软件理论基础与实践

Sub: Subtyping

胡振江 熊英飞
北京大学



子类

- 假设我们要处理如下两种记录类型
 - Person = {name:String, age:Nat}
 - Student = {name:String, age:Nat, gpa:Nat}
- 但如下项在STLC中是类型不正确的
 - ($\lambda r:\text{Person}. (r.\text{age})+1$) {name="Pat",age=21,gpa=1}
- 能否将面向对象语言中常有的子类概念引入STLC?



子类关系

- $S <: T$
 - 表示S是T的子类
- 使用子类关系

$$\frac{\text{Gamma} \vdash t_1 \in T_1 \quad T_1 <: T_2}{\text{Gamma} \vdash t_1 \in T_2} \text{(T_Sub)}$$



Nominal vs Structural

- 传统面向对象语言是名义类型系统
 - 类型之间的子类关系是用户定义的
- 在STLC中，我们希望实现结构类型系统
 - 类型之间的子类关系是系统自动推出的
- 结构类型系统常用于函数语言中
 - ML, Ocaml, Haskell
- 现代面向对象语言通常会对某些特定类型采用结构类型系统
 - 匿名函数
 - 多态/泛型



结构类型系统

- 如何判断子类关系?
 - Student是Person的子类吗?
 - Student->Student是Person->Person的子类吗?
 - Ref Student是Ref Person的子类吗?
- Liskov替换原则
 - 如果在使用T类型的值的场合，都可以替换成S类型的值，那么S就是T的子类



定义子类：传递和自反

$$\frac{S <: U \quad U <: T}{S <: T} \text{ (S_Trans)}$$

$$\overline{T <: T} \text{ (S_Refl)}$$



定义子类： Pairs

$$\frac{S_1 <: T_1 \quad S_2 <: T_2}{S_1 * S_2 <: T_1 * T_2} \quad (\text{S_Prod})$$



定义子类：Record（多条）

$$\frac{n > m}{\{i_1:T_1 \dots i_n:T_n\} <: \{i_1:T_1 \dots i_m:T_m\}} \quad (\text{S_RcdWidth})$$

$$\frac{S_1 <: T_1 \dots S_n <: T_n}{\{i_1:S_1 \dots i_n:S_n\} <: \{i_1:T_1 \dots i_n:T_n\}} \quad (\text{S_RcdDepth})$$

$$\frac{\{i_1:S_1 \dots i_n:S_n\} \text{ is a permutation of } \{j_1:T_1 \dots j_n:T_n\}}{\{i_1:S_1 \dots i_n:S_n\} <: \{j_1:T_1 \dots j_n:T_n\}} \quad (\text{S_RcdPerm})$$



定义子类： Record（单条）

$$\frac{\begin{array}{l} \text{forall } jk \text{ in } j_1..j_n, \\ \text{exists } ip \text{ in } i_1..i_m, \text{ such that} \\ jk=ip \text{ and } S_p <: T_k \end{array}}{\{i_1:S_1 \dots i_m:S_m\} <: \{j_1:T_1 \dots j_n:T_n\}} \quad (\text{S_Rcd})$$



练习

- 请写出Sum类型的子类规则

- $$\frac{S_1 <: T_1 \quad S_2 <: T_2}{S_1 + S_2 <: T_1 + T_2}$$

- 能不能加上如下规则?

$$\frac{}{T_1 * T_2 <: T_2 * T_1}$$

- 请写出List类型的子类规则

- $$\frac{S <: T}{\text{List } S <: \text{List } T}$$



定义子类：函数

- Student- \rightarrow Student是Person- \rightarrow Person的子类吗?
- 假设
 - f: Person- \rightarrow Person
 - g: Student- \rightarrow Student
 - s: Person
- 则有
 - f s类型正确
 - g s类型不正确
- g不是f的子类



定义子类：函数

- 子类关系默认考虑的是类型用作输出的情况，即为提供值而存在的
- 函数类型中还有输入类型，即为提供容器而存在的
- 整体的子类关系应该和输入类型的子类关系相反
 - 这种子部分和整体子类关系相反的形式称作逆变式 (contravariance)
 - 对应地，子部分和整体子类关系相同的形式成为协变式 (covariance)

$$\frac{T_1 <: S_1 \quad S_2 <: T_2}{S_1 \rightarrow S_2 <: T_1 \rightarrow T_2} \quad (\text{S_Arrow})$$



定义子类： 引用

- 引用既可以作为输入类型也可以作为输出类型
 - 输入类型： $a := 1$
 - 输出类型： $f(!a)$
- 引用既是逆变也是协变，即不变 (invariant)

$$\frac{S_1 <: T_1 \quad T_1 <: S_1}{\text{Ref } S_1 <: \text{Ref } T_1}$$



定义子类: Top

- 引入Top类型作为所有类型的父类
 - 类似Java中的Object

$$\frac{}{S <: \text{Top}} \quad (\text{S_Top})$$



练习(small_large_1, _2)

- 分别使得下面推导式成立的最小类型T是什么？
最大类型T是什么？

$\text{empty} \vdash (\lambda p:T \times \text{Top}. p.\text{fst}) ((\lambda z:A. z), \text{unit}) \ \text{in } A \rightarrow A$

$\text{empty} \vdash (\lambda p:(A \rightarrow A \times B \rightarrow B). p) ((\lambda z:A. z), (\lambda z:B. z)) \ \text{in } T$



练习 (count_supertypes)

- $\{x:A, y:C \rightarrow C\}$ 有多少个父类型?

Coq定义：语法



- 简单起见，只考虑STLC的子集

```
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
| tm_unit : tm
```

```
Inductive ty : Type :=
| Ty_Top : ty
| Ty_Bool : ty
| Ty_Base : string -> ty
| Ty_Arrow : ty -> ty -> ty
| Ty_Unit : ty
```

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

能否写出具有base类型的项?



Coq定义：子类关系

```
Inductive subtype : ty -> ty -> Prop :=
  | S_Refl : forall T,
    T <: T
  | S_Trans : forall S U T,
    S <: U ->
    U <: T ->
    S <: T
  | S_Top : forall S,
    S <: <{Top}>
  | S_Arrow : forall S1 S2 T1 T2,
    T1 <: S1 ->
    S2 <: T2 ->
    <{S1->S2}> <: <{T1->T2}>
where "T '<:' U" := (subtype T U).
```



Coq定义： 类型推导

```
Inductive has_type : context -> tm -> ty -> Prop :=
  (* Same as before: *)
  (* pure STLC *)
  | 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
  | T_True : forall Gamma,
    Gamma |- true \in Bool
```



Coq定义： 类型推导

```
| 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  
| T_Unit : forall Gamma,  
  Gamma |- unit \in Unit  
(* New rule of subsumption: *)  
| T_Sub : forall Gamma t1 T1 T2,  
  Gamma |- t1 \in T1 ->  
  T1 <: T2 ->  
  Gamma |- t1 \in T2
```



Progress

Theorem progress : forall t T,
empty |- t \in T ->
value t \/\ exists t', t --> t'.

- 证明概要：
 - 之前我们证明Progress的方式是在类型规则上做归纳，每条规则对应唯一的形式
 - 当前处理和之前类似，除了三种例外情况：
 - T_Sub: 直接用归纳假设可以证明
 - T_App: 我们可以知道term的形式为t1 t2且t1的类型为函数类型，但不知道t1是lambda抽象。因此，我们根据t1是否为值分情况讨论，并根据value的定义证明为值的时候一定是lambda抽象
 - T_if: 与上面情况类似，我们需要证明if条件只可能为true或者false



Preservation

Theorem `preservation` : forall t t' T,
empty |- t \in T ->
t --> t' ->
empty |- t' \in T.

- 同之前类似，在类型规则上做归纳，主要区别如下：
 - 对于T_Sub的情况，基于归纳假设可以证明
 - 之前函数调用t1 t2的证明依赖替换引理

Lemma `substitution_preserves_tpin`

`g` : forall Gamma x U t v T,
x |-> U ; Gamma |- t \in T ->
empty |- v \in U ->
Gamma |- [x:=v]t \in T.

- 但现在替换引理无法直接用，因为t1的类型并不一定是从T_Abs推出来的，也就是说不知道第一个条件是否成立



额外证明引理

```
Lemma abs_arrow : forall x S1 s2 T1 T2,  
  empty |- \x:S1,s2 \in (T1->T2) ->  
    T1 <: S1  
  /\ (x |-> S1 ; empty) |- s2 \in T2.
```

- 基于该引理，我们可以继续采用替换引理证明函数调用的情况



类型检查

- 目前的推导规则会给每个项推导出多个父类
- 实际这样实现效率较低
- 解决方案：
 - 只对每个项推导一个最小的类型
 - 函数调用的时候检查形参和实参是否有子类关系
- 具体做法：
 - 去掉T_Sub规则，采用之前的规则进行类型推导
 - 函数调用的时候，按如下规则检查实参类型S是否为形参类型T的子类
 - 如果T为Top，返回True
 - 如果T为函数类型，按函数类型递归检查
 - 如果T为Record类型，按Record类型规则递归检查



作业

- 请采用最新版英文教材
 - subtype_instances_tf_2
 - subtype_concepts_tf
 - small_large_4
 - sub_inversion_arrow
 - variations