软件理论基础与实践

REFERENCES: Typing Mutable References

胡振江 熊英飞
北京大学

引用和赋值

- STLC目前并不支持赋值语句
- 本讲给STLC加入引用和赋值语句
- 变量vs引用
  - 变量：无法通过函数参数传递
  - 引用：可以通过函数参数传递
  - IMP中没有函数调用，所以支持变量即可
- 分配一块内存
  - let a = ref 5 in
- 读取一块内存
  - !a
- 改写一块内存
  - a := 6

使用引用

- 如下程序执行结果是多少?

```
let r = ref 5 in
let s = r in
s := 82;
(!r)+1
```

使用引用：模拟封装

- 面向对象语言中，对象的内部实现对外不可见，仅能通过接口函数操作

```
newcounter =  \_:Unit.
        let c = ref 0 in
        let incc = \_:Unit. (c := succ (!c); !c) in
        let decc = \_:Unit. (c := pred (!c); !c) in
        {i=incc, d=decc}
```

- 如下代码返回什么？

```
let c1 = newcounter unit in
let c2 = newcounter unit in
let r1 = c1.i unit in
let r2 = c2.i unit in
r2  // yields 1, not 2!
```

使用引用：保存函数

```
newarray = \_:Unit. ref (\n:Nat.0)
lookup = \a:NatArray. \n:Nat. (!a) n
update = \a:NatArray. \m:Nat. \v:Nat.
              let oldf = !a in
              a := (\n:Nat. if equal m n then v else oldf n);
```

下面的update函数行为相同吗?

```
update = \a:NatArray. \m:Nat. \v:Nat.
              a := (\n:Nat. if equal m n then v else (!a) n)
```

定义引用

- ref 5约简之后的值是什么？
- 如果直接将ref tm定义为值，则会出现两个完全相同的值（如：ref 5）不相等的情况
- 引入location表示地址
  - location直接建模成整数

类型和项

```
T ::= Nat
    | Unit
    | T → T
    | Ref T
```

```
t ::= ...                Terms
    | ref t                 allocation
    | !t                    dereference
    | t := t                assignment
    | l                     location
```

Coq定义

```
Inductive ty : Type :=
  | Ty_Nat   : ty
  | Ty_Unit  : ty
  | Ty_Arrow : ty -> ty -> ty
  | Ty_Ref   : ty -> ty.

Inductive tm  : Type :=
  ……
  (* New terms: *)
  | tm_unit   : tm
  | tm_ref    : tm -> tm
  | tm_deref  : tm -> tm
  | tm_assign : tm -> tm -> tm
  | tm_loc    : nat -> tm.
```

语法解析

```
Notation "{ x }" := x (in custom stlc at level 0, x constr).

……
Notation "'Natural'" := Ty_Nat (in custom stlc at level 0).
Notation "'Ref' t" :=
  (Ty_Ref t) (in custom stlc at level 4).
Notation "'loc' x" := (tm_loc x) (in custom stlc at level 2).
Notation "'ref' x" := (tm_ref x) (in custom stlc at level 2).
Notation "'!' x " := (tm_deref x) (in custom stlc at level 2).
Notation " e1 ':=' e2 " := (tm_assign e1 e2) (in custom stlc at
 level 21).
```

修改值的定义

```
Inductive value : tm -> Prop :=
  | v_abs : forall x T2 t1,
      value <{\x:T2, t1}>
  | v_nat : forall n : nat ,
      value <{ n }>
  | v_unit :
      value <{ unit }>
  | v_loc : forall l,
      value <{ loc l }>.

Hint Constructors value : core.
```

修改替换函数的定义

```
Fixpoint subst (x : string) (s : tm) (t : tm) : tm :=
  match t with
  ……
  (* references *)
  | <{ ref t1 }> =>
      <{ ref ([x:=s] t1) }>
  | <{ !t1 }> =>
      <{ !([x:=s] t1) }>
  | <{ t1 := t2 }> =>
    <{ ([x:=s] t1) := ([x:=s] t2) }>
  | <{ loc _ }> =>
      t
  end
```

Sequence作为函数定义和调用

- 如下两项等价
  - r:=succ(!r); !r
  - (\x:Unit. !r) (r:=succ(!r)).
- 在Coq中定义

```
Definition tseq t1 t2 :=
  <{ (\ x : Unit, t2)  t1 }>.

Notation "t1 ; t2" := (tseq t1 t2) (in custom stlc at level 3).
```

定义内存空间

```
Definition store := list tm.

Definition store_lookup (n:nat) (st:store) :=
  nth n st <{ unit }>.

Fixpoint replace {A:Type} (n:nat) (x:A) (l:list A) : list A :=
  match l with
  | nil    => nil
  | h :: t =>
    match n with
    | O    => x :: t
    | S n' => h :: replace n' x t
    end
  end.
```

一些关于内存空间的引理

```
Lemma replace_nil : forall A n (x:A),
  replace n x nil = nil.

Lemma length_replace : forall A n x (l:list A),
  length (replace n x l) = length l.

Lemma lookup_replace_eq : forall l t st,
  l < length st ->
  store_lookup l (replace l t st) = t.

Lemma lookup_replace_neq : forall l1 l2 t st,
  l1 <> l2 ->
  store_lookup l1 (replace l2 t st) = store_lookup l1 st.
```

小步法操作语义：函数调用

$$\frac{\texttt{value } v_2}{(\backslash x{:}T_2.\,t1)\ v_2\ /\ st \longrightarrow [x{:=}v_2]t_1\ /\ st}\quad \text{(ST\_AppAbs)}$$

$$\frac{t_1\ /\ st \longrightarrow t_1'\ /\ st'}{t_1\ t_2\ /\ st \longrightarrow t_1'\ t_2\ /\ st'}\quad \text{(ST\_App1)}$$

$$\frac{\texttt{value } v_1 \qquad t_2\ /\ st \longrightarrow t_2'\ /\ st'}{v_1\ t_2\ /\ st \longrightarrow v_1\ t_2'\ /\ st'}\quad \text{(ST\_App2)}$$

小步法操作语义：解引用

$$\frac{t_1 \;/\; st \longrightarrow t_1' \;/\; st'}{!t_1 \;/\; st \longrightarrow !t_1' \;/\; st'} \quad \text{(ST\_Deref)}$$

$$\frac{l < |st|}{!(\mathtt{loc}\ l) \;/\; st \longrightarrow \mathtt{lookup}\ l\ st \;/\; st} \quad \text{(ST\_DerefLoc)}$$

小步法操作语义：赋值

$$\frac{t_1 \ / \ st \longrightarrow t_1' \ / \ st'}{t_1 := t_2 \ / \ st \longrightarrow t_1' := t_2 \ / \ st'} \quad \text{(ST\_Assign1)}$$

$$\frac{t_2 \ / \ st \longrightarrow t_2' \ / \ st'}{v_1 := t_2 \ / \ st \longrightarrow v_1 := t_2' \ / \ st'} \quad \text{(ST\_Assign2)}$$

$$\frac{l < |st|}{\text{loc } l := v \ / \ st \longrightarrow \text{unit} \ / \ [l:=v]st} \quad \text{(ST\_Assign)}$$

小步法操作语义：分配

$$\frac{t_1 \;/\; st \rightarrow t_1' \;/\; st'}{\text{ref } t_1 \;/\; st \rightarrow \text{ref } t_1' \;/\; st'} \quad \text{(ST\_Ref)}$$

$$\frac{}{\text{ref } v \;/\; st \rightarrow \text{loc } |st| \;/\; st, v} \quad \text{(ST\_RefValue)}$$

怎么知道一个地址的类型?

- 地址的类型取决于该地址中保存的值的类型
- 下面的类型推导规则有什么问题?

$$\frac{\texttt{Gamma; st} \vdash \texttt{lookup l st : } T_1}{\texttt{Gamma; st} \vdash \texttt{loc l : Ref } T_1}$$

- 类型推导取决于当前内存空间状态，无法静态进行
- 就算获得st，某些情况下也无法推导
  - st = [\x:Nat. (!(loc 1)) x, \x:Nat. (!(loc 0)) x]
  - 什么程序执行过程中内存空间状态会变成如上st?

解决方法：不试图知道地址类型

- loc在程序代码中不会出现，只在求值时会出现
- 静态类型检查无需知道loc的类型

$$\frac{\text{Gamma} \vdash t_1 : T_1}{\text{Gamma} \vdash \text{ref } t_1 : \text{Ref } T_1} \quad \text{(T\_Ref)}$$

$$\frac{\text{Gamma} \vdash t_1 : \text{Ref } T_1}{\text{Gamma} \vdash !t_1 : T_1} \quad \text{(T\_Deref)}$$

$$\frac{\text{Gamma} \vdash t_1 : \text{Ref } T_2 \qquad \text{Gamma} \vdash t_2 : T_2}{\text{Gamma} \vdash t_1 := t_2 : \text{Unit}} \quad \text{(T\_Assign)}$$

进展性和保持性：问题

- 在求值的过程中会产生loc
- 无法定义和证明progress和preservation

解决办法：定义地址类型

```
Definition store_ty := list ty.
Definition store_Tlookup (n:nat) (ST:store_ty) :=
  nth n ST <{ Unit }>.
```

$$\frac{l < |ST|}{\text{Gamma; ST} \vdash \text{loc } l : \text{Ref (lookup } l \text{ ST)}} \quad \text{(T\_Loc)}$$

$$\frac{\text{Gamma; ST} \vdash t_1 : T_1}{\text{Gamma; ST} \vdash \text{ref } t_1 : \text{Ref } T_1} \quad \text{(T\_Ref)}$$

$$\frac{\text{Gamma; ST} \vdash t_1 : \text{Ref } T_1}{\text{Gamma; ST} \vdash !t_1 : T_1} \quad \text{(T\_Deref)}$$

$$\frac{\begin{array}{c}\text{Gamma; ST} \vdash t_1 : \text{Ref } T_2 \\ \text{Gamma; ST} \vdash t_2 : T_2\end{array}}{\text{Gamma; ST} \vdash t_1 := t_2 : \text{Unit}} \quad \text{(T\_Assign)}$$

声明进展性和保持性

```
Definition store_well_typed (ST:store_ty) (st:store) :=
  length ST = length st /\
  (forall l, l < length st ->
     empty; ST |- { store_lookup l st } \in {store_Tlookup l ST }).
```

```
Inductive extends : store_ty -> store_ty -> Prop :=
  | extends_nil  : forall ST',
      extends ST' nil
  | extends_cons : forall x ST' ST,
      extends ST' ST ->
      extends (x::ST') (x::ST).
```

声明进展性和保持性

```
Theorem preservation : forall ST t t' T st st',
  empty ; ST |- t \in T ->
  store_well_typed ST st ->
  t / st --> t' / st' ->
  exists ST',
     extends ST' ST /\
     empty ; ST' |- t' \in T /\
     store_well_typed ST' st'.
```

```
Theorem progress : forall ST t T st,
  empty ; ST |- t \in T ->
  store_well_typed ST st ->
  (value t \/ exists t' st', t / st --> t' / st').
```

如下保持性定义有什么问题?

```
Theorem preservation_wrong1 : forall ST T t st t' st',
  empty ; ST |- t \in T ->
  t / st --> t' / st' ->
  empty ; ST |- t' \in T.
```

```
Theorem preservation_wrong2 : forall ST T t st t' st',
  empty ; ST |- t \in T ->
  t / st --> t' / st' ->
  store_well_typed ST st ->
  empty ; ST |- t' \in T.
```

证明Preservation:
适配弱化引理和替换引理

```
Lemma weakening_empty : forall Gamma ST t T,
     empty ; ST |- t \in T  ->
     Gamma ; ST |- t \in T.


Lemma substitution_preserves_typing :
  forall Gamma ST x U t v T,
  (update Gamma x U); ST |- t \in T ->
  empty ; ST |- v \in U   ->
  Gamma ; ST |- [x:=v]t \in T.
```

证明Preservation: 赋值保持类型

```
Lemma assign_pres_store_typing : forall ST st l t,
  l < length st ->
  store_well_typed ST st ->
  empty ; ST |- t \in {store_Tlookup l ST} ->
  store_well_typed ST (replace l t st).
```

- 证明： store_well_typed包括ST和st长度不变和对任意l’，st和ST的l’位置都类型正确
  - 长度不变通过替换的性质证明
  - l=l’的时候，通过前提证明
  - l≠l’的时候，通过l位置对应内容不变证明

证明Preservation: 内存空间引理

```
Lemma store_weakening : forall Gamma ST ST' t T,
  extends ST' ST ->
  Gamma ; ST |- t \in T ->
  Gamma ; ST' |- t \in T.
```

证明：在类型推导上做归纳

```
Lemma store_well_typed_app : forall ST st t1 T1,
  store_well_typed ST st ->
  empty ; ST |- t1 \in T1 ->
  store_well_typed (ST ++ T1::nil) (st ++ t1::nil).
```

证明：分成1落在ST范围内和T1范围内两种情况讨论

证明Preservation

```
Theorem preservation : forall ST t t' T st st',
  empty ; ST |- t \in T ->
  store_well_typed ST st ->
  t / st --> t' / st' ->
  exists ST',
     extends ST' ST /\
     empty ; ST' |- t' \in T /\
     store_well_typed ST' st'.
```

- 在ST和t的类型推导关系上做归纳，并根据约简关系分别讨论
  - 对于不改变st的情况，令ST’为ST，采用原preservation证明即可
  - 对于项内部有推导的情况，用归纳假设中的ST’可证

证明Preservation

```
Theorem preservation : forall ST t t' T st st',
  empty ; ST |- t \in T ->
  store_well_typed ST st ->
  t / st --> t' / st' ->
  exists ST',
     extends ST' ST /\
     empty ; ST' |- t' \in T /\
     store_well_typed ST' st'.
```

```
                l < |st|
------------------------------------------------  (ST_Assign)
loc l := v / st --> unit / [l:=v]st
```

对于ST_Assign，显然替换前后类型都是unit，
令ST’为ST，利用引理证明store_well_typed仍然保持

证明Preservation

```
Theorem preservation : forall ST t t' T st st',
  empty ; ST |- t \in T ->
  store_well_typed ST st ->
  t / st --> t' / st' ->
  exists ST',
     extends ST' ST /\
     empty ; ST' |- t' \in T /\
     store_well_typed ST' st'.
```

```
                    l < |st|
-----------------------------------------------  (ST_DerefLoc)
!(loc l) / st --> lookup l st / st
```

对于ST_DerefLoc，令ST’为ST，易证t类型不变

证明Preservation

```
Theorem preservation : forall ST t t' T st st',
  empty ; ST |- t \in T ->
  store_well_typed ST st ->
  t / st --> t' / st' ->
  exists ST',
     extends ST' ST /\
     empty ; ST' |- t' \in T /\
     store_well_typed ST' st'.
```

$$\frac{}{\texttt{ref v / st --> loc |st| / st, v}} \quad \text{(ST\_RefValue)}$$

对于ST_RefValue，令ST’为ST++V::nil，其中V是ref v中v的类型

- extends根据定义可得
- 类型不变应用对应类型推导规则可得
- store_well_typed根据引理可得

用引用支持递归

- 下面的函数r递归调用自己

```
(\r:Ref (Unit -> Unit).
    r := (\x:Unit.(!r) unit); (!r) unit)
(ref (\x:Unit.unit))
```

- 能否用这个方式写出下面的递归函数

```
factorial = fix (\n: Nat->Nat, \n : Nat,
    if iszero n then 1 else n * (factorial (n-1)))
```

答案

```
 (\r:Ref (Nat -> Nat).
        r := (\n:Nat. if iszero n then 1 else n * ((!r) (n-1)));
(!r))
   (ref (\n:Nat.0))
```

作业

- 无（基本都在课堂上讲了）