北京大學
PEKING UNIVERSITY

软件科学基础第二次习题课

孔启皓

Peking University

May 14, 2026

Table of Contents

回顾结构归纳法

```
Inductive tree (X:Type) : Type :=
    | leaf (x : X)
    | node (t1 t2 : tree X).

Check tree_ind :
    forall (X : Type) (P : tree X -> Prop),
    (forall x : X, P (leaf X x)) ->
    (forall t1 : tree X,
    P t1 -> forall t2 : tree X, P t2 -> P (node X t1 t2)) ->
    forall t : tree X, P t.
```

直观理解：一个命题对叶子成立，且两个子树成立可以推出整个树成立，则命题对所有树成立。

回顾结构归纳法

结构归纳法能够用已有的符号定义吗?

回顾结构归纳法

结构归纳法能够用已有的符号定义吗?

```
Definition nat_ind_explicit : forall P, P 0 ->
  (forall n : nat, P n -> P (S n)) ->
  forall n : nat, P n :=
    fun P => fun P0 => fun PS =>
      fix F (n:nat) : P n :=
        match n with
        | 0 => P0
        | S k => PS k (F k)
        end.
```

回顾结构归纳法

上述结构归纳法定义中使用了 Rocq 的什么特性?

回顾结构归纳法

上述结构归纳法定义中使用了 Rocq 的什么特性?

使用 `fix` 定义递归函数。

回顾结构归纳法

上述结构归纳法定义中使用了 Rocq 的什么特性?

使用 `fix` 定义递归函数。

Rocq 中的 `fix` 定义的函数必须满足什么条件?

结构递归

fix 是否能定义如下函数?

```
Definition ind_rev : forall P,
  (forall n : nat, P (S n) -> P n) ->
  forall n : nat, P n :=
    fun P => fun PS =>
      fix F (n:nat) : P n := PS n (F (S n)).
```

结构递归

```
Error:
Recursive definition of F is ill-formed.
In environment
P : nat -> Type
PS : forall n : nat, P (S n) -> P n
F : forall n : nat, P n
n : nat
Recursive call to F has principal argument equal to
"S n" instead of a subterm of "n".
Recursive definition is: "fun n : nat => PS n (F (S n))".
```

结构递归

```
Error:
Recursive definition of F is ill-formed.
In environment
P : nat -> Type
PS : forall n : nat, P (S n) -> P n
F : forall n : nat, P n
n : nat
Recursive call to F has principal argument equal to
"S n" instead of a subterm of "n".
Recursive definition is: "fun n : nat => PS n (F (S n))".
```

什么是"ill-formed"?

结构递归

检查 Fix 的类型：

```
Fix
  : forall (A : Type) (R : A -> A -> Prop),
    well_founded R ->
    forall P : A -> Type,
    (forall x : A, (forall y : A, R y x -> P y) -> P x) ->
    forall x : A, P x

well_founded =
fun (A : Type) (R : A -> A -> Prop) => forall a : A, Acc R a
     : forall [A : Type], (A -> A -> Prop) -> Prop

Inductive Acc (A : Type) (R : A -> A -> Prop) (x : A) : Prop :=
  Acc_intro : (forall y : A, R y x -> Acc R y) -> Acc R x.
```

结构递归

`Acc` 在数学中被称为"accessibility"，它的定义表明了一个元素 x 往更小的元素 y 递归是安全的。假如我们构造出了 `Acc R x`，也就说明了从 x 出发不存在无穷下降的序列。这也正是 well-founded 的定义。

对于一般的递归函数，根据子结构的定义可以天然证明 well-founded 关系，因此 Coq 可以接受它们的定义。

结构递归

使用 Fix 定义 nat_ind：

```
Definition nat_ind_by_Fix (P : nat -> Prop) (H0 : P 0)
  (HS : forall n : nat, P n -> P (S n)) :
    forall n : nat, P n := Fix lt_wf
      (fun n => P n)
      (fun n rec =>
         match n with
         | 0 => fun _ => H0
         | S n' => fun rec' => HS n' (rec' n' (lt_succ_diag_r n'))
         end rec).
```

其中 lt_wf 是 lt 的 well-founded 证明。lt_succ_diag_r 是 $n < Sn$ 的证明。

霍尔逻辑复习

while 的霍尔逻辑规则是什么？

霍尔逻辑复习

while 的霍尔逻辑规则是什么?

```
Theorem hoare_while : forall P (b:bexp) c,
  {{P /\ b}} c {{P}} ->
  {{P}} while b do c end {{P /\ ~ b}}.
```

wp 复习

skip，Assign，Seq，If 的最弱前条件定义是什么？

wp 复习

skip，Assign，Seq，If 的最弱前条件定义是什么？

```
wp(skip, Q) = Q
wp(x := a, Q) = Q[x |-> a]
wp(c1; c2, Q) = wp(c1, wp(c2, Q))
wp(if b then c1 else c2 end, Q) =
  (b -> wp(c1, Q)) /\ (~b -> wp(c2, Q))
```

wp 复习

skip，Assign，Seq，If 的最弱前条件定义是什么？

```
wp(skip, Q) = Q
wp(x := a, Q) = Q[x |-> a]
wp(c1; c2, Q) = wp(c1, wp(c2, Q))
wp(if b then c1 else c2 end, Q) =
  (b -> wp(c1, Q)) /\ (~b -> wp(c2, Q))
```

除了 while 以外，其他命令的 wp 都可以直接递归定义。而 while 的 wp 则需要使用量词定义。

wp 复习

skip，Assign，Seq，If 的最弱前条件定义是什么？

```
wp(skip, Q) = Q
wp(x := a, Q) = Q[x |-> a]
wp(c1; c2, Q) = wp(c1, wp(c2, Q))
wp(if b then c1 else c2 end, Q) =
  (b -> wp(c1, Q)) /\ (~b -> wp(c2, Q))
```

除了 while 以外，其他命令的 wp 都可以直接递归定义。而 while 的 wp 则需要使用量词定义。
一个有意思问题：while 的 wp 唯一吗？

循环不变式

当我们已知后条件，同时又不需要严格最弱前条件时，就可以用循环不变式来证明这个前条件。

循环不变式的条件?

循环不变式

当我们已知后条件，同时又不需要严格最弱前条件时，就可以用循环不变式来证明这个前条件。

循环不变式的条件？足够弱，足够强，能保持。

一些找循环不变式技巧：

- 不会修改的变量总是在不变式中。
- 完整代入前条件进行一轮推导，找变量间的递减关系。
- 没啥技巧了，直接理解程序然后瞪不变式。

自动程序验证

Rocq 中自动证明工具效果如何?

```
Definition sqrt_dec (m:nat) : decorated :=
  <{
    {{ X = m }} ->> {{ X = m }} Z := 0 {{ Z = 0 /\ X = m }};
      while ((Z+1)*(Z+1) <= X) do {{ ... }}
        Z := Z + 1 {{ Z * Z <= X /\ X = m }}
      end
    {{ ... }} ->> {{ Z*Z<=m /\ m<(Z+1)*(Z+1) }}
  }>.

Theorem sqrt_correct : forall m, outer_triple_valid (sqrt_dec m).
Proof. verify. Qed.
```

自动了，但还是要提供很多信息。

自动程序验证

我们已经拥有了 wp 这个强大 (?) 的工具，能否直接用它来验证程序?

自动程序验证

我们已经拥有了 wp 这个强大 (?) 的工具，能否直接用它来验证程序？
当程序中没有 while 时，直接对后条件 Post 不断应用用 wp，即可导出一个不包含量词的前条件 Pre'。只需证明 Pre -> Pre' 就可以了。而这个命题中仅包含逻辑表达式和算术表达式，可以直接调用 SMT 求解器来验证。
例如：

```
// pre: 0 <= x
if (x >= 10) x = 10; else x = x + 1;
// post: 1 <= x <= 10
```

就可以直接用 wp 转换成命题：

$$(0 \le x) \to ((x \ge 10 \to 1 \le 10 \le 10) \land (x < 10 \to 1 \le x + 1 \le 10))$$

自动程序验证

当程序中有 while 时，wp 的定义中就包含了量词，（可能）无法直接调用 SMT 求解器来验证。此时我们可以引入循环不变式来消除量词。
SMT 求解器：可以理解为一个黑盒工具，当输入是仅包含逻辑与（简单）算术表达式的命题时，大概率能自动判断其真值。
这实际是目前自动程序验证工具的主流做法：先调用 SMT 求解器来验证，当 SMT 求解器超时或无法求解时，让用户提供循环不变式。

Frama-C

Frama-C 是一个针对 C 语言的自动程序验证工具，支持多种验证方法，包括基于 wp 的方法。用户可以在 C 代码中使用 ACSL（ANSI/ISO C Specification Language）来编写规范，然后 Frama-C 会尝试自动验证这些规范。

相比于 IMP 语言，C 语言复杂得多，包含了指针、动态内存分配、函数调用等特性。因此 Frama-C 的验证比 IMP 更加复杂。

Frama-C

一个例子：

```
/*@
  requires x <= 10;
  ensures \result == 10;
*/
int inc_to_ten(int x) {
  /*@
    loop invariant x <= 10;
    loop assigns x;
    loop variant 10 - x;
  */
  while (x < 10) { x++; }
  return x;
}
```

Frama-C

一个更复杂的例子：

```
31   /*@
32     requires n > 0;
33     requires \valid(a + (0 .. n-1));
34     assigns \nothing;
35     ensures \forall integer k; 0 <= k < n ==> \result >= a[k];
36     ensures \exists integer k; 0 <= k < n && \result == a[k];
37   */
38 int func(int *a, int n)
39 {
40   int max = a[0];
41   int i   = 1;
42
43   /*@
44     loop invariant prefix_max(a, i, n, max);
45     loop assigns i, max;
46     loop variant  n - i;
47   */
48   while (i < n) {
49     if (a[i] > max) {
50       max = a[i];
51     }
52     i++;
53   }
54
55   return max;
56 }
```

Frama-C

实际上 wp 仅是 Frama-C 中的一个插件，Frama-C 还提供了其他插件，如 Eva 用于静态分析，E-ACSL 用于运行时验证等。不过这些插件的原理和实现细节与课程内容关系不大。

总得来说，Frama-C wp 是一个功能强大的工具，但由于 C 语言本身的复杂性，使用它进行验证相比 IMP 语言要复杂得多。

VST-A

VST-A 同样是一个针对 C 语言的自动程序验证工具，基于 VST/Coq 进行注释式 C 验证。
VST-A 在 VST 的基础上进行了自动化改进。VST 是一个基于 Separation Logic 的 C 程序验证工具，在课程最后（应该）会进行一些介绍。

VST-A

VST-A 的创新之一在于让用户插入断言，将函数分解为断言间的路径。这样对于每条路径，都可以转化成几乎直线的 Hoare triple 验证问题，极大地简化了验证过程。

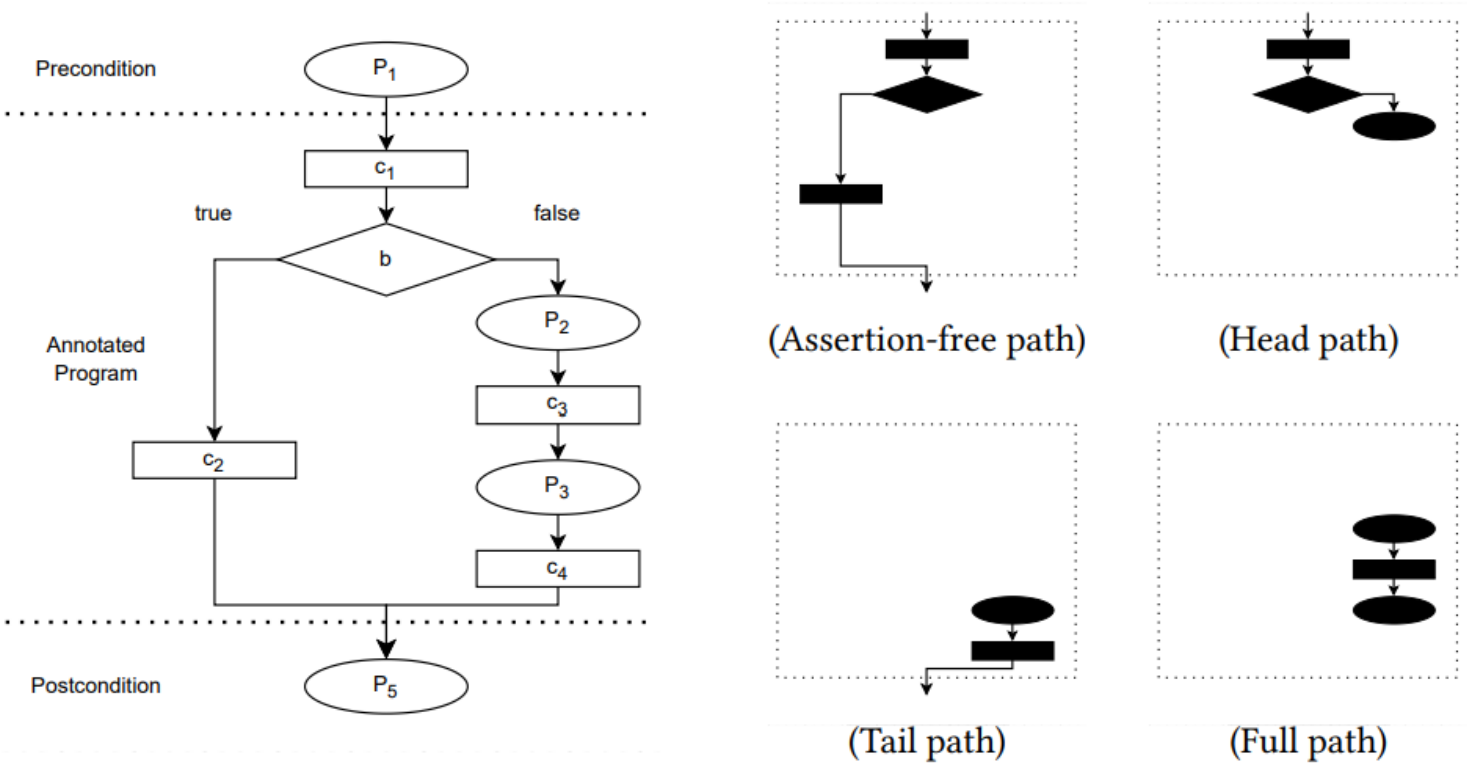

Fig. 12. Control flow graphs v.s. paths in split results

Dafny

Dafny 是一个针对 Dafny 语言的自动程序验证工具。相比 Frama-C 和 VST-A，Dafny 进行了最大程度的封装与自动化，用户几乎不需要关心（也很难关心）底层的验证细节。

```
method IncToTen(x0: int) returns (x: int)
  requires x0 <= 10
  ensures x == 10
{
  x := x0;
  while x < 10
    invariant x <= 10 // This line is unnecessary
  {
    x := x + 1;
  }
}
```

Dafny

对于不包含数据结构的简单程序，Dafny 的自动化效果非常好，用户几乎不需要提供任何信息（甚至有时不需要提供循环不变式）。但对于包含数据结构的程序，Dafny 无法自动验证，就需要用户提供验证细节了。

```
19  method DifferenceMinMax(a: seq<int>) returns (diff: int)
20      requires |a| > 0
21      ensures diff == Max(a[..]) - Min(a[..])
22  {
23      var min := a[0];
24      var max := a[0];
25
26      if |a| == 1 { diff := 0; return; }
27      var i := 1;
28      while i < |a|
29          invariant 1 <= i <= |a|
30          invariant min == Min(a[..i])
31          invariant max == Max(a[..i])
32      {
33          if a[i] <= min {
34              min := a[i];
35          }
36          if a[i] >= max {
37              max := a[i];
38          }
39          assert a[..i] + [a[i]] == a[..i+1];
40          i := i + 1;
41      }
42      assert a[..i] == a[..]; // ????
43      diff := max - min;
44  }
```

Dafny

对于更复杂的程序，例如证明 01 背包最优性，证明极长

```
149   method solve(time: seq<int>, value: seq<int>, totalTime: int)
313
314                       assert prevMaxValue[i] <= prevMaxValue[i - t] + v;
315                       assert best(prevTime, prevValue, i, prevMaxValue[i]);
316                       max_choose_equal(prevTime, prevValue, i, prevMaxValue[i], t, v);
317                       if !best(time, value, i, result) {
318                           var choose2 :| !max_possible(time, value, choose2, i, result);
319                           assert valid_subseq(|time|, choose2) &&
320                               sum_choose(time, choose2) <= i &&
321                               sum_choose(value, choose2) > result;
322                           if |choose2| > 0 && choose2[|choose2|-1] == |prevTime| {
323                               var choose2_sub := choose2[..|choose2|-1];
324                               assert max_possible(prevTime, prevValue, choose2_sub, i - t,
                                  prev);
325                               valid_subseq_equal(|time|, choose2);
326                               assert valid_subseq(|prevTime|, choose2_sub);
327                               assert sum_choose(time, choose2_sub) <= i - t;
328                               sum_choose_add(prevTime, t, choose2_sub);
329                               assert sum_choose(prevTime, choose2_sub) <= i - t;
330                               assert sum_choose(prevValue, choose2_sub) <= prev;
331                               sum_choose_add(prevValue, v, choose2_sub);
332                               assert sum_choose(value, choose2_sub) <= prev + v;
333                               assert false;
334                           } else {
335                           }
336                           assert false;
337                       }
338                   }
339                   assert best(time, value, i, result);
340               }
341               maxValue := maxValue + [result];
342               i := i + 1;
343           }
344       }
345   }
```

更好的自动化证明

既然循环不变量是自动程序验证中的一个难点，能否让 LLMs “猜”一个循环不变量，然后验证？

现在有大量的研究工作在探索这个方向，例如 Refine4LLM[1] 和 SpecGen[2] 等工具，已经在一些简单程序上取得了不错的效果。

[1] Yufan Cai, Zhe Hou, David Sanán, Xiaokun Luan, Yun Lin, Jun Sun, Jin Song Dong: Automated Program Refinement: Guide and Verify Code Large Language Model with Refinement Calculus. Proc. ACM Program. Lang. 9(POPL): 2057-2089 (2025)

[2] Lezhi Ma, Shangqing Liu, Yi Li, Xiaofei Xie, Lei Bu: SpecGen: Automated Generation of Formal Program Specifications via Large Language Models. ICSE 2025: 16-28