面向形式化证明的程序设计语言

王迪

如何开发高安全级别的软件？

- **形式化证明**是目前能提供全面软件安全保障的**唯一**技术
- 案例：严格形式化证明的操作系统微内核 **seL4**
  - 在美国军用无人机上部署，被认为是最安全的无人机
- 瓶颈一：开发门槛、成本极高
  - seL4 的证明花费 **>>20 人年**
- 瓶颈二：开发过程需掌握多种工具
  - seL4 的开发使用 C 作为**实现**语言，Haskell 作为**规约**语言，Isabelle/HOL 作为**证明**工具

如何在写代码的时候就尽量保证程序安全且正确？

CMU 15-122: Principles of **Imperative Computation**

CMU 15-122: Principles of **Imperative Computation**

- 一年级本科生的 C 语言编程课

CMU 15-122: Principles of **Imperative Computation**

- 一年级本科生的 C 语言编程课
- 学生已有基本的编程经验（比如 Python）

CMU 15-122: Principles of **Imperative Computation**

- 一年级本科生的 C 语言编程课
- 学生已有基本的编程经验（比如 Python）
- 注重讲授编写**安全**且**正确**的 C 程序的方法

CMU 15-122: Principles of **Imperative Computation**

- 一年级本科生的 C 语言编程课
- 学生已有基本的编程经验（比如 Python）
- 注重讲授编写**安全**且**正确**的 C 程序的方法
- 第一节课：**Contracts**

Contracts

```
int f(int x, int y) {
  int r = 1;
  while (y > 1) {
    if (y % 2 == 1) {
      r = x * r;
    }
    x = x * x;
    y = y / 2;
  }
  return r * x;
}
```

Contracts

```
int f(int x, int y) {
  int r = 1;
  while (y > 1) {
    if (y % 2 == 1) {
      r = x * r;
    }
    x = x * x;
    y = y / 2;
  }
  return r * x;
}
```

- 左边的代码做了什么事情?

Contracts

```c
int f(int x, int y) {
  int r = 1;
  while (y > 1) {
    if (y % 2 == 1) {
      r = x * r;
    }
    x = x * x;
    y = y / 2;
  }
  return r * x;
}
```

- 左边的代码做了什么事情？
- 这段代码有 bug 吗？

Contracts

```
int f(int x, int y) {
  int r = 1;
  while (y > 1) {
    if (y % 2 == 1) {
      r = x * r;
    }
    x = x * x;
    y = y / 2;
  }
  return r * x;
}
```

- 左边的代码做了什么事情?
- 这段代码有 bug 吗?
- 如何通过语言技术来提升代码的可信度?

Contracts

```
int f(int x, int y) {
  int r = 1;
  while (y > 1) {
    if (y % 2 == 1) {
      r = x * r;
    }
    x = x * x;
    y = y / 2;
  }
  return r * x;
}
```

- 左边的代码做了什么事情？
- 这段代码有 bug 吗？
- 如何通过语言技术来提升代码的可信度？
- 通过 **Contracts** 为代码添加信息

语言技术能提供什么？

语言技术能提供什么？

B

- 1969 年
- 无类型系统
- 性能差，不安全

语言技术能提供什么？

B → C

- 1969 年
- 无类型系统
- 性能差，不安全

- 1972 年
- 简单类型系统
- 性能好，基本安全

语言技术能提供什么？

B →	C →	Rust
◉ 1969 年	◉ 1972 年	◉ 2015 年
◉ 无类型系统	◉ 简单类型系统	◉ 复杂类型系统
◉ 性能差，不安全	◉ 性能好，基本安全	◉ 性能不错，更加安全

语言技术能提供什么？

B → C → Rust

- 1969 年
- 无类型系统
- 性能差，不安全

- 1972 年
- 简单类型系统
- 性能好，基本安全

- 2015 年
- 复杂类型系统
- 性能不错，更加安全

语言技术能提供什么？

B → C → Rust

- 1969 年
- 无类型系统
- 性能差，不安全

- 1972 年
- 简单类型系统
- 性能好，基本安全

- 2015 年
- 复杂类型系统
- 性能不错，更加安全

No Free Lunch：程序员需要对代码进行更多的设计和分析

语言技术能提供什么？

```
int f(int x, int y)
//@require y >= 0;
//@ensure \result == POW(x, y);
{
  int r = 1, b = x, e = y;
  while (e > 1)
  //@loop_invariant e >= 0;
  //@loop_invariant POW(b,e) * r == POW(x,y);
  {
    if (e % 2 == 1) {
      r = b * r;
    }
    b = b * b;
    e = e / 2;
  }
  //@assert e == 0;
  return r;
}
```

语言技术能提供什么？

```
int f(int x, int y)
//@require y >= 0;
//@ensure \result == POW(x, y);
{
  int r = 1, b = x, e = y;
  while (e > 1)
  //@loop_invariant e >= 0;
  //@loop_invariant POW(b,e) * r == POW(x,y);
  {
    if (e % 2 == 1) {
      r = b * r;
    }
    b = b * b;
    e = e / 2;
  }
  //@assert e == 0;
  return r;
}
```

- **核心：** 把动态检查 contracts 的过程静态化

语言技术能提供什么？

```
int f(int x, int y)
//@require y >= 0;
//@ensure \result == POW(x, y);
{
  int r = 1, b = x, e = y;
  while (e > 1)
  //@loop_invariant e >= 0;
  //@loop_invariant POW(b,e) * r == POW(x,y);
  {
    if (e % 2 == 1) {
      r = b * r;
    }
    b = b * b;
    e = e / 2;
  }
  //@assert e == 0;
  return r;
}
```

- **核心**：把动态检查 contracts 的过程静态化
- **No Free Lunch**：程序员需要自己标注前后条件、循环不变式以及断言

语言技术能提供什么？

```c
int f(int x, int y)
//@require y >= 0;
//@ensure \result == POW(x, y);
{
  int r = 1, b = x, e = y;
  while (e > 1)
  //@loop_invariant e >= 0;
  //@loop_invariant POW(b,e) * r == POW(x,y);
  {
    if (e % 2 == 1) {
      r = b * r;
    }
    b = b * b;
    e = e / 2;
  }
  //@assert e == 0;
  return r;
}
```

- **核心：** 把动态检查 contracts 的过程静态化
- **No Free Lunch：** 程序员需要自己标注前后条件、循环不变式以及断言
- **自动化：** 尽可能减少程序员需要写的东西

语言技术能提供什么？

```
int f(int x, int y)
//@require y >= 0;
//@ensure \result == POW(x, y);
{
  int r = 1, b = x, e = y;
  while (e > 1)
  //@loop_invariant e >= 0;
  //@loop_invariant POW(b,e) * r == POW(x,y);
  {
    if (e % 2 == 1) {
      r = b * r;
    }
    b = b * b;
    e = e / 2;
  }
  //@assert e == 0;
  return r;
}
```

自动分析循环体
达成的效果
（符号执行）

- **核心**：把动态检查 contracts 的过程静态化
- **No Free Lunch**：程序员需要自己标注前后条件、循环不变式以及断言
- **自动化**：尽可能减少程序员需要写的东西

语言技术：把 Contracts 静态化

```
int f(int x, int y)
//@require y >= 0;
//@ensure \result == POW(x, y);
{
  int r = 1, b = x, e = y;
  while (e > 1)
  //@loop_invariant e >= 0;
  //@loop_invariant POW(b,e) * r == POW(x,y);
  {
    if (e % 2 == 1) {
      r = b * r;
    }
    b = b * b;
    e = e / 2;
  }
  //@assert e == 0;
  return r;
}
```

语言技术：把 Contracts 静态化

```
int f(int x, int y)
//@require y >= 0;
//@ensure \result == POW(x, y);
{
  int r = 1, b = x, e = y;
  while (e > 1)
  //@loop_invariant e >= 0;
  //@loop_invariant POW(b,e) * r == POW(x,y);
  {
    if (e % 2 == 1) {
      r = b * r;
    }
    b = b * b;
    e = e / 2;
  }
  //@assert e == 0;
  return r;
}
```

- 如何静态检查 contracts？

语言技术：把 Contracts 静态化

```
int f(int x, int y)
//@require y >= 0;
//@ensure \result == POW(x, y);
{
  int r = 1, b = x, e = y;
  while (e > 1)
  //@loop_invariant e >= 0;
  //@loop_invariant POW(b,e) * r == POW(x,y);
  {
    if (e % 2 == 1) {
      r = b * r;
    }
    b = b * b;
    e = e / 2;
  }
  //@assert e == 0;
  return r;
}
```

- 如何静态检查 contracts？
- 分为两部分：

语言技术：把 Contracts 静态化

```
int f(int x, int y)
//@require y >= 0;
//@ensure \result == POW(x, y);
{
  int r = 1, b = x, e = y;
  while (e > 1)
  //@loop_invariant e >= 0;
  //@loop_invariant POW(b,e) * r == POW(x,y);
  {
    if (e % 2 == 1) {
      r = b * r;
    }
    b = b * b;
    e = e / 2;
  }
  //@assert e == 0;
  return r;
}
```

- 如何静态检查 contracts？
- 分为两部分：
  - 静态分析代码的效果

语言技术：把 Contracts 静态化

```
int f(int x, int y)
//@require y >= 0;
//@ensure \result == POW(x, y);
{
  int r = 1, b = x, e = y;
  while (e > 1)
  //@loop_invariant e >= 0;
  //@loop_invariant POW(b,e) * r == POW(x,y);
  {
    if (e % 2 == 1) {
      r = b * r;
    }
    b = b * b;
    e = e / 2;
  }
  //@assert e == 0;
  return r;
}
```

- 如何静态检查 contracts？
- 分为两部分：
  - 静态分析代码的效果
  - 证明性质之间的关系

静态分析代码的效果

```
{
  if (e % 2 == 1) {
    r = b * r;
  }
  b = b * b;
  e = e / 2;
}
```

静态分析代码的效果

```
{
  if (e % 2 == 1) {
    r = b * r;
  }
  b = b * b;
  e = e / 2;
}
```

```
r=r0, b=b0, e=e0
```

静态分析代码的效果

```
{
  if (e % 2 == 1) {
    r = b * r;
  }
  b = b * b;
  e = e / 2;
}
```

```
r=r0, b=b0, e=e0
```

↓

```
r = (e0%2==1) ? b0*r0 : r0
b = b0*b0
e = e0/2
```

静态分析代码的效果

```
{
  if (e % 2 == 1) {
    r = b * r;
  }
  b = b * b;
  e = e / 2;
}
```

r=r0, b=b0, e=e0

↓ 通过符号执行自动实现

r = (e0%2==1) ? b0*r0 : r0
b = b0*b0
e = e0/2

证明性质之间的关系

```
//@loop_invariant
  POW(b,e) * r == POW(x,y);
{
  if (e % 2 == 1) {
    r = b * r;
  }
  b = b * b;
  e = e / 2;
}
```

```
r=r0, b=b0, e=e0
```

```
r = (e0%2==1) ? b0*r0 : r0
b = b0*b0
e = e0/2
```

证明性质之间的关系

```
//@loop_invariant
  POW(b,e) * r == POW(x,y);
{
  if (e % 2 == 1) {
    r = b * r;
  }
  b = b * b;
  e = e / 2;
}
```

```
r=r0, b=b0, e=e0

    POW(b0,e0)*r0 == POW(x,y)
==> POW(b,e)*r == POW(x,y)

r = (e0%2==1) ? b0*r0 : r0
b = b0*b0
e = e0/2
```

证明性质之间的关系

```
//@loop_invariant
  POW(b,e) * r == POW(x,y);
{
  if (e % 2 == 1) {
    r = b * r;
  }
  b = b * b;
  e = e / 2;
}
```

r=r0, b=b0, e=e0

- 采取命令式的交互式定理证明

```
    POW(b0,e0)*r0 == POW(x,y)
==> POW(b,e)*r == POW(x,y)
```

```
r = (e0%2==1) ? b0*r0 : r0
b = b0*b0
e = e0/2
```

证明性质之间的关系

```
//@loop_invariant
  POW(b,e) * r == POW(x,y);
{
  if (e % 2 == 1) {
    r = b * r;
  }
  b = b * b;
  e = e / 2;
}
```

```
r=r0, b=b0, e=e0
```

- 采取命令式的交互式定理证明
- 采取基于 SMT 等的自动证明技术

```
    POW(b0,e0)*r0 == POW(x,y)
==> POW(b,e)*r == POW(x,y)

r = (e0%2==1) ? b0*r0 : r0
b = b0*b0
e = e0/2
```

C*：开发与证明一体化的系统级编程语言

功能代码

规约代码

证明代码

C*：开发与证明一体化的系统级编程语言

功能代码	规约代码	证明代码
普通 C 代码		

C*：开发与证明一体化的系统级编程语言

功能代码	规约代码	证明代码
普通 C 代码	前、后条件 循环不变式 断言、规约函数	

C*：开发与证明一体化的系统级编程语言

功能代码	规约代码	证明代码
普通 C 代码	前、后条件 循环不变式 断言、规约函数	证明规约之间的关系

C*：开发与证明一体化的系统级编程语言

功能代码	规约代码	证明代码
普通 C 代码	前、后条件 循环不变式 断言、规约函数	证明规约之间的关系

命令式风格

C*：开发与证明一体化的系统级编程语言

功能代码	规约代码	证明代码
普通 C 代码	前、后条件 循环不变式 断言、规约函数	证明规约之间的关系

命令式风格

代码对程序状态进行操作

C*：开发与证明一体化的系统级编程语言

功能代码

普通 C 代码

规约代码

前、后条件
循环不变式
断言、规约函数

证明代码

证明规约之间的关系

命令式风格

代码对程序状态进行操作
规约对程序状态进行描述

C*：开发与证明一体化的系统级编程语言

功能代码	规约代码	证明代码
普通C代码	前、后条件 循环不变式 断言、规约函数	证明规约之间的关系

命令式风格

代码对程序状态进行操作
规约对程序状态进行描述
证明对证明状态进行操作

如何使用命令式的语言设计使得形式化证明对 C 程序员友好？

如何写证明？

```
int POW(int x, int y)
//@require y >= 0;
{
  if (y == 0) return 1;
  return POW(x, y-1)*x;
}
```

如何写证明？

```
int POW(int x, int y)
//@require y >= 0;
{
  if (y == 0) return 1;
  return POW(x, y-1)*x;
}
```

```
thm POW_x_zero_is_one(int x)
//@prove POW(x, 0) == 1;
{
  …
}
```

如何写证明？

证明也是一种计算，它的结果是一个成立的命题

```
int POW(int x, int y)
//@require y >= 0;
{
  if (y == 0) return 1;
  return POW(x, y-1)*x;
}
```

```
thm POW_x_zero_is_one(int x)
//@prove POW(x, 0) == 1;
{
  …
}
```

如何写证明？

证明也是一种计算，它的结果是一个成立的命题

```
int POW(int x, int y)
//@require y >= 0;
{
  if (y == 0) return 1;
  return POW(x, y-1)*x;
}
```

```
thm POW_x_zero_is_one(int x)
//@prove POW(x, 0) == 1;
{
  thm l1 = refl(`POW(x,0)`);
  // l1: |- POW(x,0)==POW(x,0);
  …
}
```

如何写证明？

证明也是一种计算，它的结果是一个成立的命题

证明的中间状态是一些成立的命题

```
int POW(int x, int y)
//@require y >= 0;
{
  if (y == 0) return 1;
  return POW(x, y-1)*x;
}
```

```
thm POW_x_zero_is_one(int x)
//@prove POW(x, 0) == 1;
{
  thm l1 = refl(`POW(x,0)`);
  // l1: |- POW(x,0)==POW(x,0);
  …
}
```

如何写证明？

证明也是一种计算，它的结果是一个成立的命题

证明的中间状态是一些成立的命题

```
int POW(int x, int y)
//@require y >= 0;
{
  if (y == 0) return 1;
  return POW(x, y-1)*x;
}
```

```
thm POW_x_zero_is_one(int x)
//@prove POW(x, 0) == 1;
{
  thm l1 = refl(`POW(x,0)`);
  // l1: |- POW(x,0)==POW(x,0);
  thm l2 = rewrite(l1, `POW(x,0)`);
  // l2: |- POW(x,0) == { if (0==0) return 1;
                          return POW(x,0-1)*x; };
  …
}
```

如何写证明？

证明也是一种计算，它的结果是一个成立的命题

证明的中间状态是一些成立的命题

```
int POW(int x, int y)
//@require y >= 0;
{
  if (y == 0) return 1;
  return POW(x, y-1)*x;
}
```

```
thm POW_x_zero_is_one(int x)
//@prove POW(x, 0) == 1;
{
  thm l1 = refl(`POW(x,0)`);
  // l1: |- POW(x,0)==POW(x,0);
  thm l2 = rewrite(l1, `POW(x,0)`);
  // l2: |- POW(x,0) == { if (0==0) return 1;
                          return POW(x,0-1)*x; };

  …
}
```

这些断言可看做**证明**层面的状态断言

如何写证明?

证明也是一种计算，它的结果是一个成立的命题

证明的中间状态是一些成立的命题

```
int POW(int x, int y)
//@require y >= 0;
{
  if (y == 0) return 1;
  return POW(x, y-1)*x;
}
```

```
thm POW_x_zero_is_one(int x)
//@prove POW(x, 0) == 1;
{
  thm l1 = refl(`POW(x,0)`);
  // l1: |- POW(x,0)==POW(x,0);
  thm l2 = rewrite(l1, `POW(x,0)`);
  // l2: |- POW(x,0) == { if (0==0) return 1;
                           return POW(x,0-1)*x; };
  …
}
```

证明层面的函数描述了逻辑推理的规则，用来构造新的命题

这些断言可看做**证明**层面的状态断言

基于 LCF 的交互式定理证明

基于 LCF 的交互式定理证明

- 采取与 Isabelle/HOL 相同的证明核心设计，但直接使用类 C 语言来操作证明对象

基于 LCF 的交互式定理证明

- 采取与 Isabelle/HOL 相同的证明核心设计，但直接使用类 C 语言来操作证明对象
- 可把定理视作**抽象数据结构**，证明核心提供了合法操作该结构的 API

基于 LCF 的交互式定理证明

- 采取与 Isabelle/HOL 相同的证明核心设计，但直接使用类 C 语言来操作证明对象
- 可把定理视作**抽象数据结构**，证明核心提供了合法操作该结构的 API
- 支持 SMT 等自动证明只需要简单扩展该 API 即可！

```
thm l1 = smt(`POW(x,0) == 1`);
// l1: |- POW(x,0)==1;
```

基于 LCF 的交互式定理证明

- 采取与 Isabelle/HOL 相同的证明核心设计，但直接使用类 C 语言来操作证明对象
- 可把定理视作**抽象数据结构**，证明核心提供了合法操作该结构的 API
- 支持 SMT 等自动证明只需要简单扩展该 API 即可！

```
thm l1 = smt(`POW(x,0) == 1`);
// l1: |- POW(x,0)==1;
```

- 问题：可以像 Coq 那样进行**后向证明**，或者 Agda 那样进行**等式证明**吗？

命令式风格的后向证明

命令式风格的后向证明

- **后向证明：** 从目标出发，通过 tactic 变换目标

命令式风格的后向证明

- **后向证明：** 从目标出发，通过 tactic 变换目标
- 例如要证明 A /\ B，通过 `split` tactic 可以得到两个子目标 A 和 B

命令式风格的后向证明

- **后向证明：** 从目标出发，通过 tactic 变换目标
- 例如要证明 A /\ B，通过 `split` tactic 可以得到两个子目标 A 和 B
- 现有工作已展示 LCF 框架可以支持后向证明！

命令式风格的后向证明

- **后向证明：**从目标出发，通过 tactic 变换目标
- 例如要证明 `A /\ B`，通过 `split` tactic 可以得到两个子目标 A 和 B
- 现有工作已展示 LCF 框架可以支持后向证明！

```
type goal = thm list * term
```

命令式风格的后向证明

- **后向证明：** 从目标出发，通过 tactic 变换目标
- 例如要证明 A /\ B，通过 split tactic 可以得到两个子目标 A 和 B
- 现有工作已展示 LCF 框架可以支持后向证明！

```
type goal = thm list * term
type goal_state = goal list * (thm list -> thm)
```

命令式风格的后向证明

- **后向证明：**从目标出发，通过 tactic 变换目标
- 例如要证明 `A /\ B`，通过 `split` tactic 可以得到两个子目标 A 和 B
- 现有工作已展示 LCF 框架可以支持后向证明！

```
type goal = thm list * term
type goal_state = goal list * (thm list -> thm)
type tactic = goal -> goal_state
```

命令式风格的后向证明

- 把 goal 相关类型也看做**抽象数据结构**，但**不需要**改动证明核心

命令式风格的后向证明

- 把 goal 相关类型也看做**抽象数据结构**，但**不需要**改动证明核心

```
proof pf = initialize(`1 == 1 && 0 == 0`);
// goals: [1 == 1 && 0 == 0], cont: \[thm] -> thm
```

命令式风格的后向证明

- 把 goal 相关类型也看做**抽象数据结构**，但**不需要**改动证明核心

```
proof pf = initialize(`1 == 1 && 0 == 0`);
// goals: [1 == 1 && 0 == 0], cont: \[thm] -> thm

split(pf);
// goals: [1 == 1; 0 == 0], cont: \[thm1; thm2] -> conjunct(thm1, thm2)
```

命令式风格的后向证明

- 把 goal 相关类型也看做**抽象数据结构**，但**不需要**改动证明核心

```
proof pf = initialize(`1 == 1 && 0 == 0`);
// goals: [1 == 1 && 0 == 0], cont: \[thm] -> thm

split(pf);
// goals: [1 == 1; 0 == 0], cont: \[thm1; thm2] -> conjunct(thm1, thm2)

reflexivity(pf, `1`);
// goals: [0 == 0], cont: \[thm2] -> conjunct(refl(1), thm2)
```

命令式风格的后向证明

- 把 goal 相关类型也看做**抽象数据结构**，但**不需要**改动证明核心

```
proof pf = initialize(`1 == 1 && 0 == 0`);
// goals: [1 == 1 && 0 == 0], cont: \[thm] -> thm

split(pf);
// goals: [1 == 1; 0 == 0], cont: \[thm1; thm2] -> conjunct(thm1, thm2)

reflexivity(pf, `1`);
// goals: [0 == 0], cont: \[thm2] -> conjunct(refl(1), thm2)

thm zero_eq_zero = smt(`0 == 0`);
```

命令式风格的后向证明

- 把 goal 相关类型也看做**抽象数据结构**，但**不需要**改动证明核心

```
proof pf = initialize(`1 == 1 && 0 == 0`);
// goals: [1 == 1 && 0 == 0], cont: \[thm] -> thm

split(pf);
// goals: [1 == 1; 0 == 0], cont: \[thm1; thm2] -> conjunct(thm1, thm2)

reflexivity(pf, `1`);
// goals: [0 == 0], cont: \[thm2] -> conjunct(refl(1), thm2)

thm zero_eq_zero = smt(`0 == 0`);

apply(pf, zero_eq_zero);
// goals: [], cont: \[] -> conjunct(refl(1), zero_eq_zero)
```

命令式风格的后向证明

- 把 goal 相关类型也看做**抽象数据结构**，但**不需要**改动证明核心

```
proof pf = initialize(`1 == 1 && 0 == 0`);
// goals: [1 == 1 && 0 == 0], cont: \[thm] -> thm

split(pf);
// goals: [1 == 1; 0 == 0], cont: \[thm1; thm2] -> conjunct(thm1, thm2)

reflexivity(pf, `1`);
// goals: [0 == 0], cont: \[thm2] -> conjunct(refl(1), thm2)

thm zero_eq_zero = smt(`0 == 0`);

apply(pf, zero_eq_zero);
// goals: [], cont: \[] -> conjunct(refl(1), zero_eq_zero)

thm ret = qed(pf);
```

命令式风格的等式证明

命令式风格的等式证明

- **等式证明：** 用等式把一系列 rewrite 步骤连在一起，符合人类写证明的直观

命令式风格的等式证明

- **等式证明：** 用等式把一系列 rewrite 步骤连在一起，符合人类写证明的直观
- 提供相应 API 即可，也**不需要**改动证明核心

命令式风格的等式证明

- **等式证明：** 用等式把一系列 rewrite 步骤连在一起，符合人类写证明的直观
- 提供相应 API 即可，也**不需要**改动证明核心

```
eqproof pf = start_with(`1 + 1`);
```

命令式风格的等式证明

- **等式证明：** 用等式把一系列 rewrite 步骤连在一起，符合人类写证明的直观
- 提供相应 API 即可，也**不需要**改动证明核心

```
eqproof pf = start_with(`1 + 1`);

thm thm1 = smt(`1 + 1 == 2`);
extend(pf, `2`, thm1);
// |- 1 + 1 == 2
```

命令式风格的等式证明

- **等式证明：** 用等式把一系列 rewrite 步骤连在一起，符合人类写证明的直观
- 提供相应 API 即可，也**不需要**改动证明核心

```
eqproof pf = start_with(`1 + 1`);

thm thm1 = smt(`1 + 1 == 2`);
extend(pf, `2`, thm1);
// |- 1 + 1 == 2

thm thm2 = smt(`2 == 4 / 2`);
extend(pf, `4 / 2`, thm2);
// |- 1 + 1 == 4 / 2
```

命令式风格的等式证明

- **等式证明：** 用等式把一系列 rewrite 步骤连在一起，符合人类写证明的直观

- 提供相应 API 即可，也**不需要**改动证明核心

```
eqproof pf = start_with(`1 + 1`);

thm thm1 = smt(`1 + 1 == 2`);
extend(pf, `2`, thm1);
// |- 1 + 1 == 2

thm thm2 = smt(`2 == 4 / 2`);
extend(pf, `4 / 2`, thm2);
// |- 1 + 1 == 4 / 2

thm ret = terminate(pf);
```

怎么编写程序规约呢？
怎么在证明中用规约呢？

如何写规约函数？

```
int POW(int x, int y)
//@require y >= 0;
{
  if (y == 0) return 1;
  return POW(x, y-1)*x;
}
```

如何写规约函数？

```
int POW(int x, int y)
//@require y >= 0;
{
  if (y == 0) return 1;
  return POW(x, y-1)*x;
}
```

- 「不容易写错」

如何写规约函数？

```
int POW(int x, int y)
//@require y >= 0;
{
  if (y == 0) return 1;
  return POW(x, y-1)*x;
}
```

- 「不容易写错」
- 无副作用

如何写规约函数?

```
int POW(int x, int y)
//@require y >= 0;
{
  if (y == 0) return 1;
  return POW(x, y-1)*x;
}
```

- 「不容易写错」
- 无副作用
- 但是可以用 Python 风格的数组

如何写规约函数？

```
int POW(int x, int y)
//@require y >= 0;
{
  if (y == 0) return 1;
  return POW(x, y-1)*x;
}
```

```
bool is_in(int x, int[] A, int lo, int hi)
//@require 0 <= lo && lo <= hi && hi <= \length(A);
{
  if (lo == hi) return false;
  return A[lo] == x || is_in(x, A, lo + 1, hi);
}
```

- 「不容易写错」
- 无副作用
- 但是可以用 Python 风格的数组

如何写规约函数？

```
int POW(int x, int y)
//@require y >= 0;
{
  if (y == 0) return 1;
  return POW(x, y-1)*x;
}
```

```
bool is_in(int x, int[] A, int lo, int hi)
//@require 0 <= lo && lo <= hi && hi <= \length(A);
{
  if (lo == hi) return false;
  return A[lo] == x || is_in(x, A, lo + 1, hi);
}
```

```
bool le_seg(int x, int[] A, int lo, int hi)
//@require …
{
  if (lo == hi) return true;
  return x <= A[lo] && le_seg(x, A, lo + 1, hi);
}
```

- 「不容易写错」
- 无副作用
- 但是可以用 Python 风格的数组

如何写规约函数？

```
int POW(int x, int y)
//@require y >= 0;
{
  if (y == 0) return 1;
  return POW(x, y-1)*x;
}
```

- 「不容易写错」
- 无副作用
- 但是可以用 Python 风格的数组

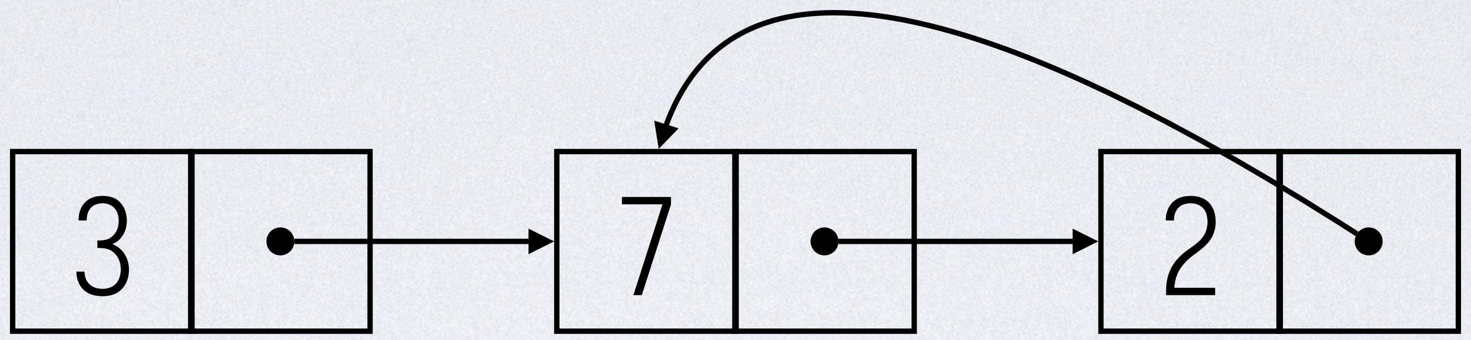
```
bool is_in(int x, int[] A, int lo, int hi)
//@require 0 <= lo && lo <= hi && hi <= \length(A);
{
  if (lo == hi) return false;
  return A[lo] == x || is_in(x, A, lo + 1, hi);
}
```

```
bool le_seg(int x, int[] A, int lo, int hi)
//@require …
{
  if (lo == hi) return true;
  return x <= A[lo] && le_seg(x, A, lo + 1, hi);
}
```

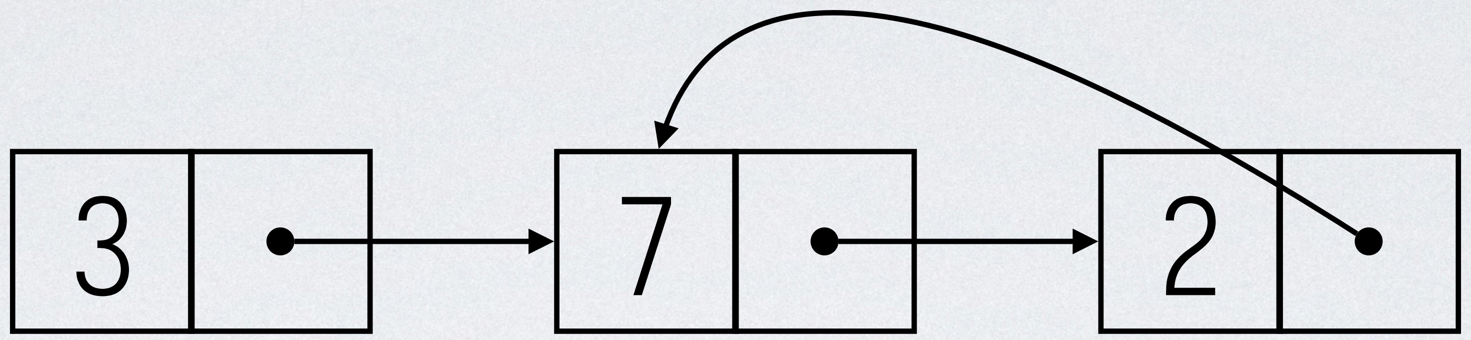
```
bool is_sorted(int[] A, int lo, int hi)
//@require …
{
  if (lo == hi) return true;
  return le_seg(A[lo], A, lo + 1, hi) && is_sorted(A, lo + 1, hi);
}
```

分离逻辑：适用于描述堆性质的规约语言

分离逻辑：适用于描述堆性质的规约语言

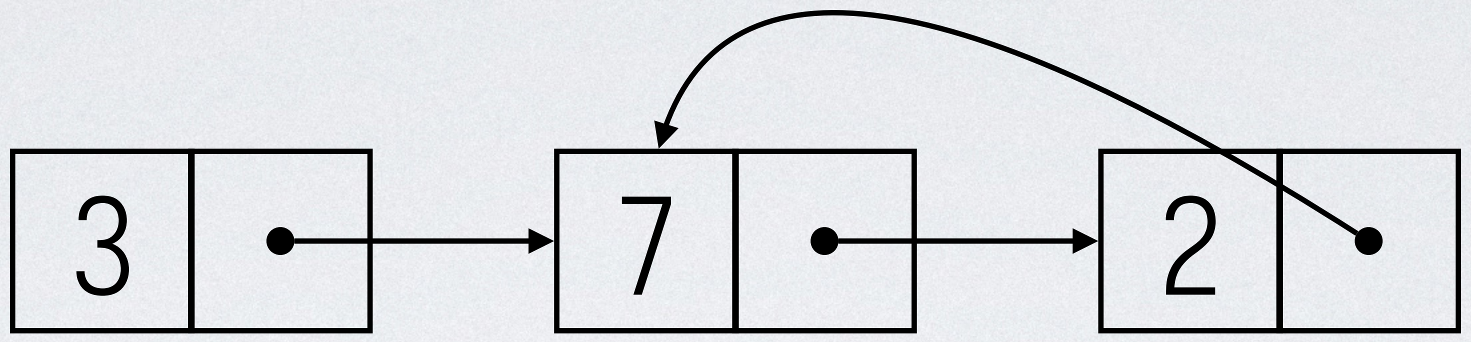


分离逻辑：适用于描述堆性质的规约语言




```
l1~>(3,l2) * l2~>(7,l3) * l3~>(2,l2)
```

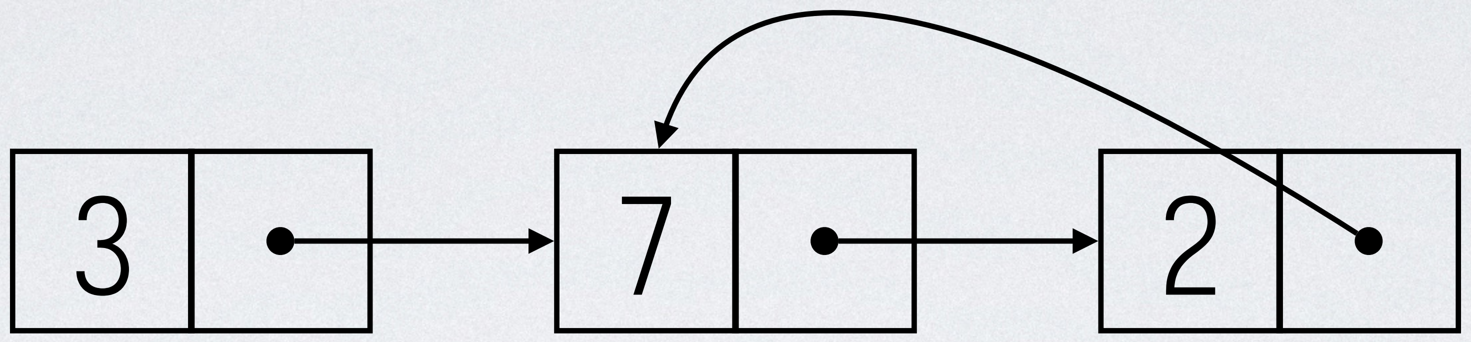
分离逻辑：适用于描述堆性质的规约语言



l1~>(3,l2) * l2~>(7,l3) * l3~>(2,l2)

loc~>data 表示一块堆空间，其中只有一个地址 loc，指向的数据为 data

分离逻辑：适用于描述堆性质的规约语言

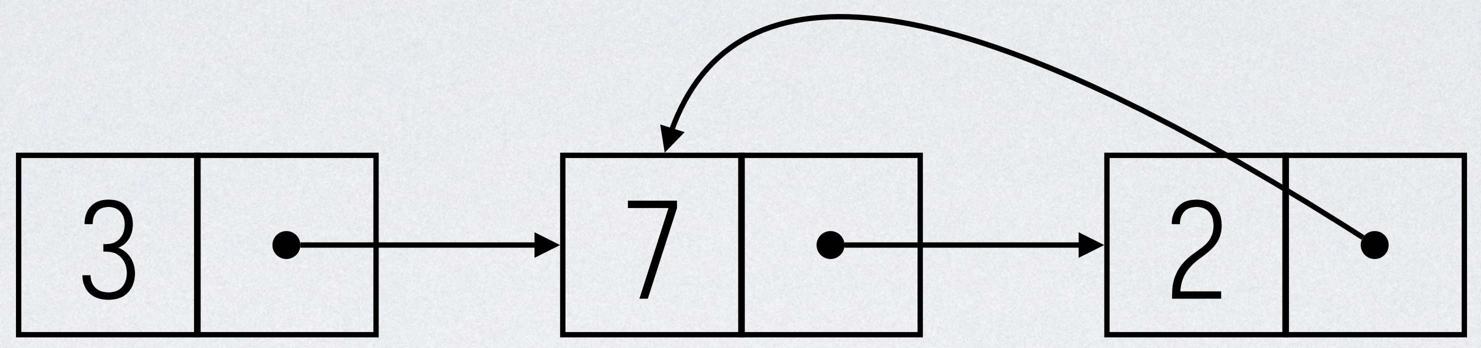


`l1~>(3,l2) * l2~>(7,l3) * l3~>(2,l2)`

loc~>data 表示一块堆空间，其中只有一个地址 loc，指向的数据为 data

* 表示合并两块地址**不相交**的堆空间

分离逻辑：适用于描述堆性质的规约语言



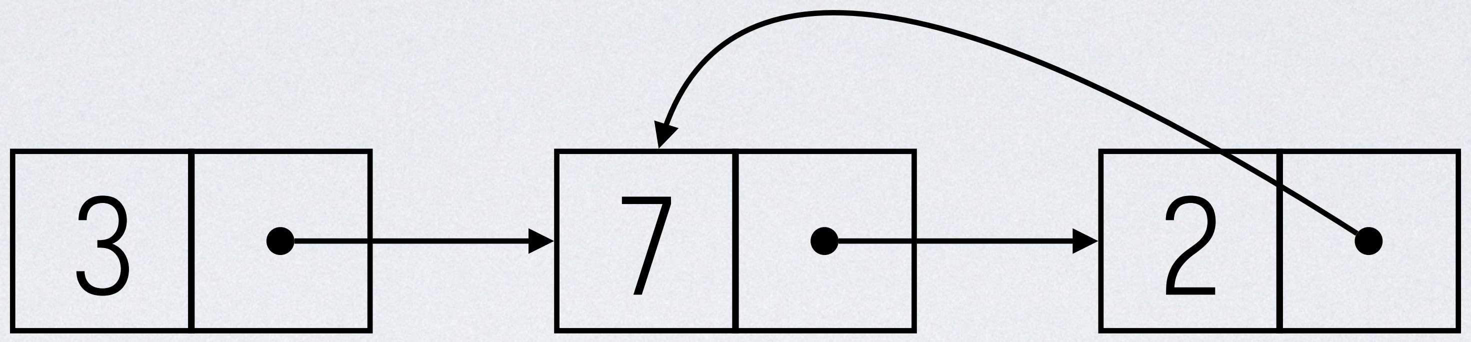
l1~>(3,l2) * l2~>(7,l3) * l3~>(2,l2)

loc~>data 表示一块堆空间，其中只有一个地址 loc，指向的数据为 data

* 表示合并两块地址**不相交**的堆空间

```
bool is_segment(heap h, list* start, list* end) {
  if (start == NULL) return false;
  if (start == end) return is_empty(h);
  return is_point_to(h, start, *start) &&
         is_segment(remove(h, start), start->next, end);
}
```

分离逻辑：适用于描述堆性质的规约语言



l1~>(3,l2) * l2~>(7,l3) * l3~>(2,l2)

loc~>data 表示一块堆空间，其中只有一个地址 loc，指向的数据为 data

* 表示合并两块地址**不相交**的堆空间

```
bool is_segment(heap h, list* start, list* end) {
  if (start == NULL) return false;
  if (start == end) return is_empty(h);
  return is_point_to(h, start, *start) &&
         is_segment(remove(h, start), start->next, end);
}
```

堆空间 h 是否恰好描述了从 start 到 end 的一段链表

分离逻辑：解耦内存布局和数据结构

分离逻辑：解耦内存布局和数据结构

- C 程序中的数据结构通常是经过优化的，有精巧设计的内存布局

分离逻辑：解耦内存布局和数据结构

- C程序中的数据结构通常是经过优化的，有精巧设计的内存布局
- 但是，在描述功能时，通常不需要知道所有的实现细节，可以进行适度抽象

分离逻辑：解耦内存布局和数据结构

- C程序中的数据结构通常是经过优化的，有精巧设计的内存布局
- 但是，在描述功能时，通常不需要知道所有的实现细节，可以进行适度抽象
  - 单/双向链表，实现了线性表结构，其功能可用**数组**描述

分离逻辑：解耦内存布局和数据结构

- C程序中的数据结构通常是经过优化的，有精巧设计的内存布局
- 但是，在描述功能时，通常不需要知道所有的实现细节，可以进行适度抽象
  - 单/双向链表，实现了线性表结构，其功能可用**数组**描述
  - 哈希表/红黑树，实现了键-值查询结构，其功能可用**映射**描述

分离逻辑：解耦内存布局和数据结构

- C程序中的数据结构通常是经过优化的，有精巧设计的内存布局
- 但是，在描述功能时，通常不需要知道所有的实现细节，可以进行适度抽象
  - 单/双向链表，实现了线性表结构，其功能可用**数组**描述
  - 哈希表/红黑树，实现了键-值查询结构，其功能可用**映射**描述

```
bool is_segment(heap h, list* start, list* end, int[] A, int lo, int hi)
//@require 0 <= lo && lo <= hi && hi <= \length(A);
{
  if (start == NULL) return false;
  if (lo == hi) return start == end && is_empty(h);
  return start->data == A[lo] &&
         is_point_to(h, start, *start) &&
         is_segment(remove(h, start), start->next, end, A, lo + 1, hi);
}
```

分离逻辑：解耦内存布局和数据结构

- C程序中的数据结构通常是经过优化的，有精巧设计的内存布局
- 但是，在描述功能时，通常不需要知道所有的实现细节，可以进行适度抽象
  - 单/双向链表，实现了线性表结构，其功能可用**数组**描述
  - 哈希表/红黑树，实现了键-值查询结构，其功能可用**映射**描述

```
bool is_segment(heap h, list* start, list* end, int[] A, int lo, int hi)
//@require 0 <= lo && lo <= hi && hi <= \length(A);
{
  if (start == NULL) return false;
  if (lo == hi) return start == end && is_empty(h);
  return start->data == A[lo] &&
         is_point_to(h, start, *start) &&
         is_segment(remove(h, start), start->next, end, A, lo + 1, hi);
}
```

堆空间h是否恰好描述了从start到end的一段链表，其正好存储了数组A[lo…hi]的数据

分离逻辑：对堆性质的声明式规约

```
bool is_segment(heap h, list* start, list* end, int[] A, int lo, int hi)
//@require 0 <= lo && lo <= hi && hi <= \length(A);
{
  if (start == NULL) return false;
  if (lo == hi) return start == end && is_empty(h);
  return start->data == A[lo] &&
         is_point_to(h, start, *start) &&
         is_segment(remove(h, start), start->next, end, A, lo + 1, hi);
}
```

分离逻辑：对堆性质的声明式规约

```
bool is_segment(heap h, list* start, list* end, int[] A, int lo, int hi)
//@require 0 <= lo && lo <= hi && hi <= \length(A);
{
  if (start == NULL) return false;
  if (lo == hi) return start == end && is_empty(h);
  return start->data == A[lo] &&
         is_point_to(h, start, *start) &&
         is_segment(remove(h, start), start->next, end, A, lo + 1, hi);
}
```

```
heap_prop is_segment(list* start, list* end, int[] A, int lo, int hi)
//@require 0 <= lo && lo <= hi && hi <= \length(A);
{
  if (start == NULL) return false /\ empty;
  if (lo == hi) return (start == end) /\ empty;
  return (start->data == A[lo]) /\
         (start ~> *start) *
         is_segment(start->next, end, A, lo + 1, hi);
}
```

分离逻辑：对堆性质的声明式规约

```
bool is_segment(heap h, list* start, list* end, int[] A, int lo, int hi)
//@require 0 <= lo && lo <= hi && hi <= \length(A);
{
  if (start == NULL) return false;
  if (lo == hi) return start == end && is_empty(h);
  return start->data == A[lo] &&
         is_point_to(h, start, *start) &&
         is_segment(remove(h, start), start->next, end, A, lo + 1, hi);
}
```

描述堆性质的谓词

```
heap_prop is_segment(list* start, list* end, int[] A, int lo, int hi)
//@require 0 <= lo && lo <= hi && hi <= \length(A);
{
  if (start == NULL) return false /\ empty;
  if (lo == hi) return (start == end) /\ empty;
  return (start->data == A[lo]) /\
         (start ~> *start) *
         is_segment(start->next, end, A, lo + 1, hi);
}
```

分离逻辑：解耦内存布局和数据结构

```
void list_rev(list* l)
//@parameter int[] A;
//@require is_segment(l, NULL, A, 0, \length(A));
//@ensure is_segment(l, NULL, rev(A, \length(A)), 0, \length(A));
{
  …
}
```

分离逻辑：解耦内存布局和数据结构

```
void list_rev(list* l)
//@parameter int[] A;
//@require is_segment(l, NULL, A, 0, \length(A));
//@ensure is_segment(l, NULL, rev(A, \length(A)), 0, \length(A));
{
  …
}
```

```
int[] rev(int[] A, int n)
//@require n == \length(A);
//@ensure n == \length(\result);
{
  int[] B = alloc_array(int, n);
  for (int i = 0; i < n; i ++) {
    B[i] = A[n - 1 - i];
  }
  return B;
}
```

分离逻辑：解耦内存布局和数据结构

```
void list_rev(list* l)
//@parameter int[] A;
//@require is_segment(l, NULL, A, 0, \length(A));
//@ensure is_segment(l, NULL, rev(A, \length(A)), 0, \length(A));
{
  …
}
```

```
int[] rev(int[] A, int n)
//@require n == \length(A);
//@ensure n == \length(\result);
{
  int[] B = alloc_array(int, n);
  for (int i = 0; i < n; i ++) {
    B[i] = A[n - 1 - i];
  }
  return B;
}
```

解耦内存布局
和计算性质

连接规约与证明

在证明时，我们需要用到什么信息？

连接规约与证明

在证明时，我们需要用到什么信息？

```
int POW(int x, int y)
//@require y >= 0;
{
  if (y == 0) return 1;
  return POW(x, y-1)*x;
}
```

连接规约与证明

在证明时，我们需要用到什么信息？

```
int POW(int x, int y)
//@require y >= 0;
{
  if (y == 0) return 1;
  return POW(x, y-1)*x;
}
```

```
y==0 ==> POW(x,y)==1

y>0  ==> POW(x,y)==POW(x,y-1)*x
```

连接规约与证明

在证明时，我们需要用到什么信息？

```
int[] rev(int[] A, int n)
//@require n == \length(A);
//@ensure n == \length(\result);
{
  int[] B = alloc_array(int, n);
  for (int i = 0; i < n; i ++) {
    B[i] = A[n - 1 - i];
  }
  return B;
}
```

连接规约与证明

在证明时，我们需要用到什么信息？

```
int[] rev(int[] A, int n)
//@require n == \length(A);
//@ensure n == \length(\result);
{
  int[] B = alloc_array(int, n);
  for (int i = 0; i < n; i ++) {
    B[i] = A[n - 1 - i];
  }
  return B;
}
```

```
n == len(A) ==>
  rev(A, n) == rev_loop(A, array(n), n, 0)

i < n       ==>
  rev_loop(A, B, n, i) ==
  rev_loop(A, put(B,i,get(A,n-1-i)), n, i + 1)

i == n      ==>
  rev_loop(A, B, n, i) == B
```

连接规约与证明

```
int POW(int x, int y)
//@require y >= 0;
{
  if (y == 0) return 1;
  return POW(x, y-1)*x;
}
```

```
y==0 ==> POW(x,y)==1
y>0  ==> POW(x,y)==POW(x,y-1)*x
```

```
int[] rev(int[] A, int n)
//@require n == \length(A);
//@ensure n == \length(\result);
{
  int[] B = alloc_array(int, n);
  for (int i = 0; i < n; i ++) {
    B[i] = A[n - 1 - i];
  }
  return B;
}
```

```
n == len(A) ==>
  rev(A, n) == rev_loop(A, array(n), n, 0)
i < n       ==>
  rev_loop(A, B, n, i) ==
  rev_loop(A, put(B,i,get(A,n-1-i)), n, i + 1)
i == n      ==>
  rev_loop(A, B, n, i) == B
```

连接规约与证明

```
int POW(int x, int y)
//@require y >= 0;
{
  if (y == 0) return 1;
  return POW(x, y-1)*x;
}
```

```
y==0 ==> POW(x,y)==1
y>0  ==> POW(x,y)==POW(x,y-1)*x
```

```
int[] rev(int[] A, int n)
//@require n == \length(A);
//@ensure n == \length(\result);
{
  int[] B = alloc_array(int, n);
  for (int i = 0; i < n; i ++) {
    B[i] = A[n - 1 - i];
  }
  return B;
}
```

方案：利用关于规约函数满足的等式来进行性质的证明

```
n == len(A) ==>
  rev(A, n) == rev_loop(A, array(n), n, 0)
i < n       ==>
  rev_loop(A, B, n, i) ==
  rev_loop(A, put(B,i,get(A,n-1-i)), n, i + 1)
i == n      ==>
  rev_loop(A, B, n, i) == B
```

连接规约与证明

```
int POW(int x, int y)
//@require y >= 0;
{
  if (y == 0) return 1;
  return POW(x, y-1)*x;
}
```

```
y==0 ==> POW(x,y)==1
y>0  ==> POW(x,y)==POW(x,y-1)*x
```

连接规约与证明

```
int POW(int x, int y)
//@require y >= 0;
{
  if (y == 0) return 1;
  return POW(x, y-1)*x;
}
```

```
thm POW_rule1(int x, int y);
//@prove y == 0 ==> POW(x,y)==1;

thm POW_rule2(int x, int y);
//@prove y > 0 ==> POW(x,y)==POW(x,y-1)*x;
```

```
y==0 ==> POW(x,y)==1
y>0  ==> POW(x,y)==POW(x,y-1)*x
```

连接规约与证明

```
int POW(int x, int y)
//@require y >= 0;
{
  if (y == 0) return 1;
  return POW(x, y-1)*x;
}
```

```
y==0 ==> POW(x,y)==1
y>0  ==> POW(x,y)==POW(x,y-1)*x
```

```
thm POW_rule1(int x, int y);
//@prove y == 0 ==> POW(x,y)==1;

thm POW_rule2(int x, int y);
//@prove y > 0 ==> POW(x,y)==POW(x,y-1)*x;
```

```
thm POW_x_one_is_x(int x)
//@prove POW(x, 1) == x;
{
  thm l1 = POW_rule2(x, 1);
  // l1: |- POW(x,1)==POW(x,1-1)*x;
  thm l2 = POW_rule1(x, 0);
  // l2: |- POW(x,0)==1;
  thm l3 = trans(l1, l2);
  // l3: |- POW(x,1)==1*x;
  return l3;
}
```

短期原型实现方案：基于 VST-A

C*

功能代码	规约代码	证明代码

短期原型实现方案：基于 VST-A

C*

功能代码

规约代码

证明代码

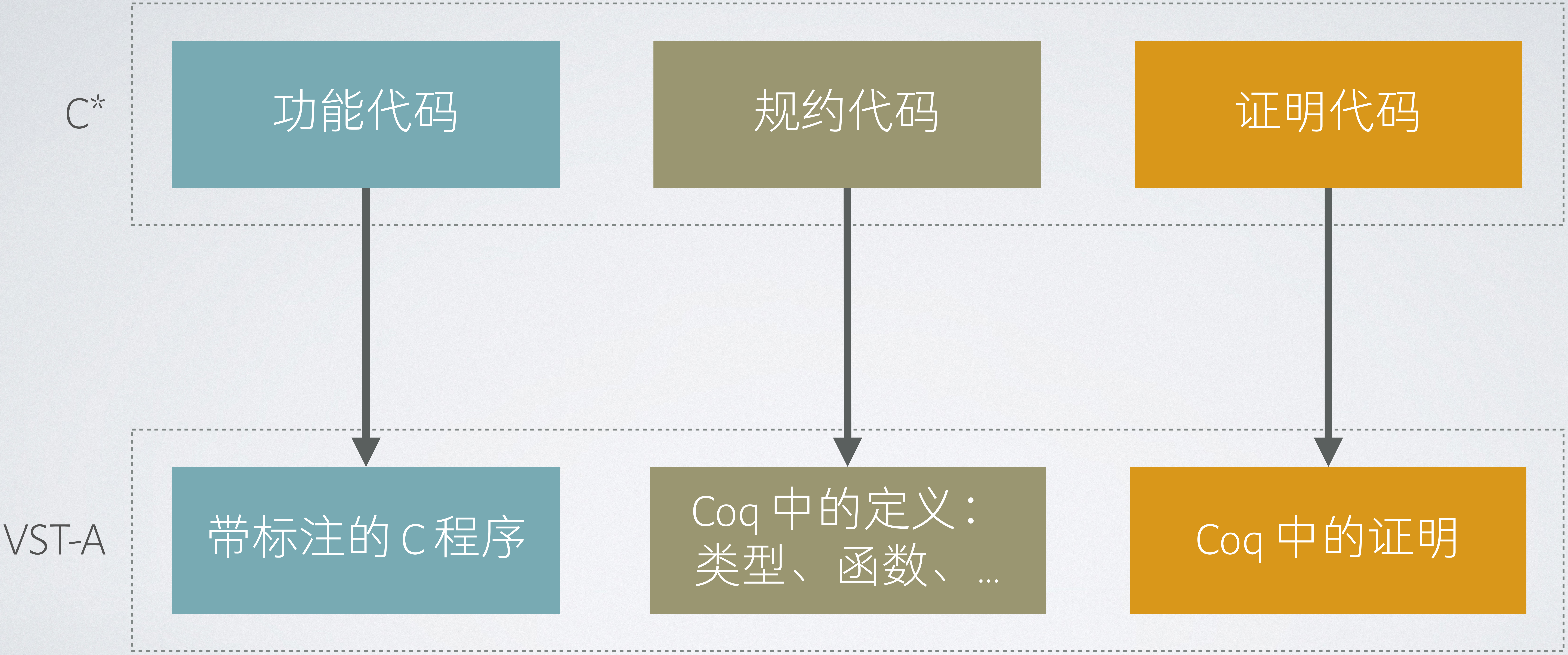
VST-A

带标注的 C 程序

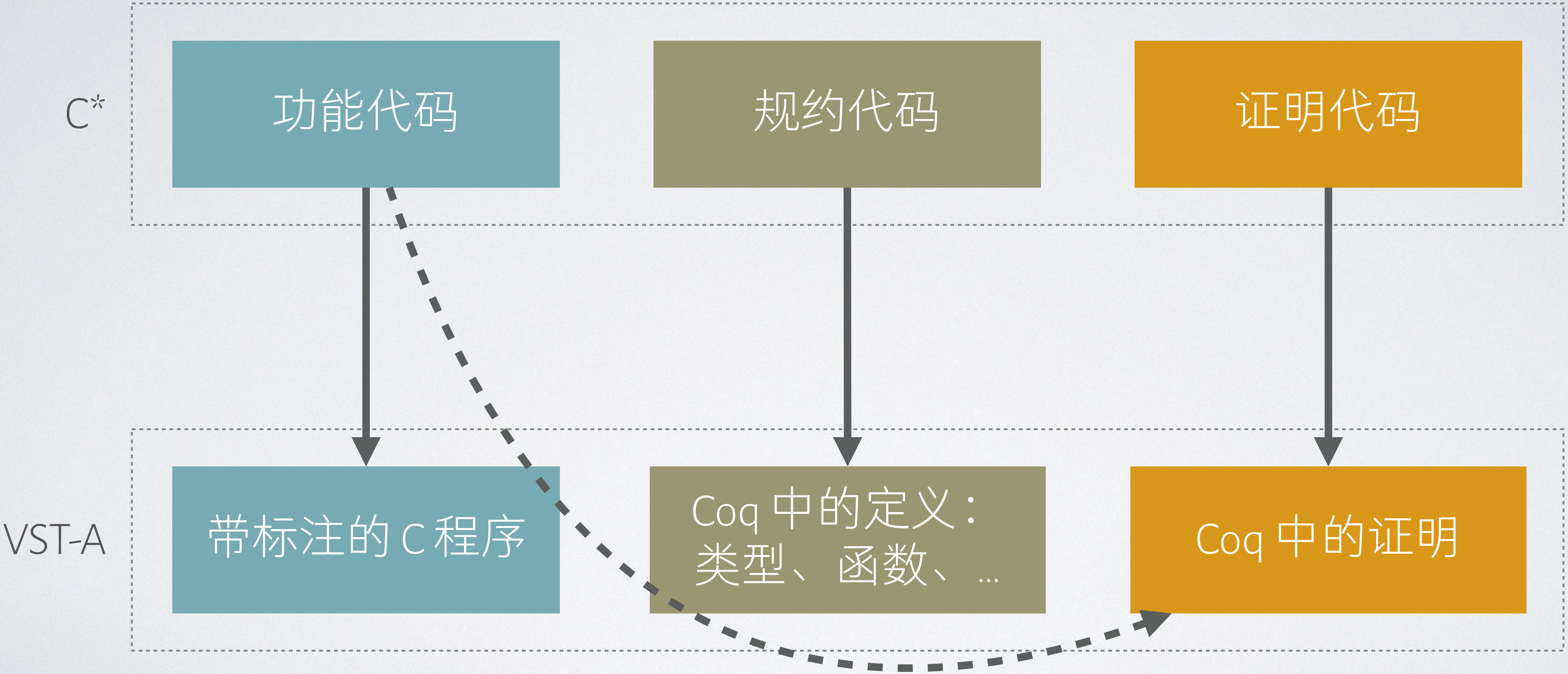
Coq 中的定义：
类型、函数、...

Coq 中的证明

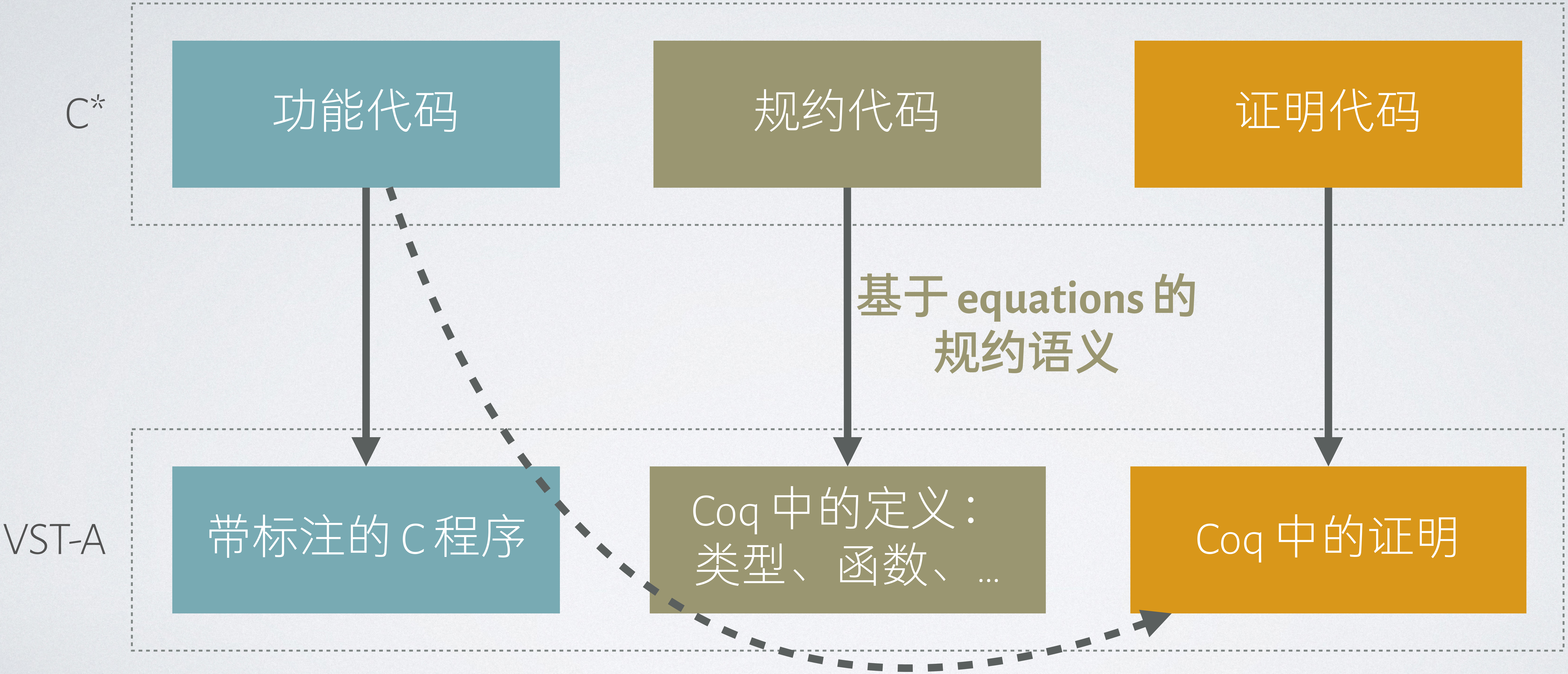
短期原型实现方案：基于 VST-A



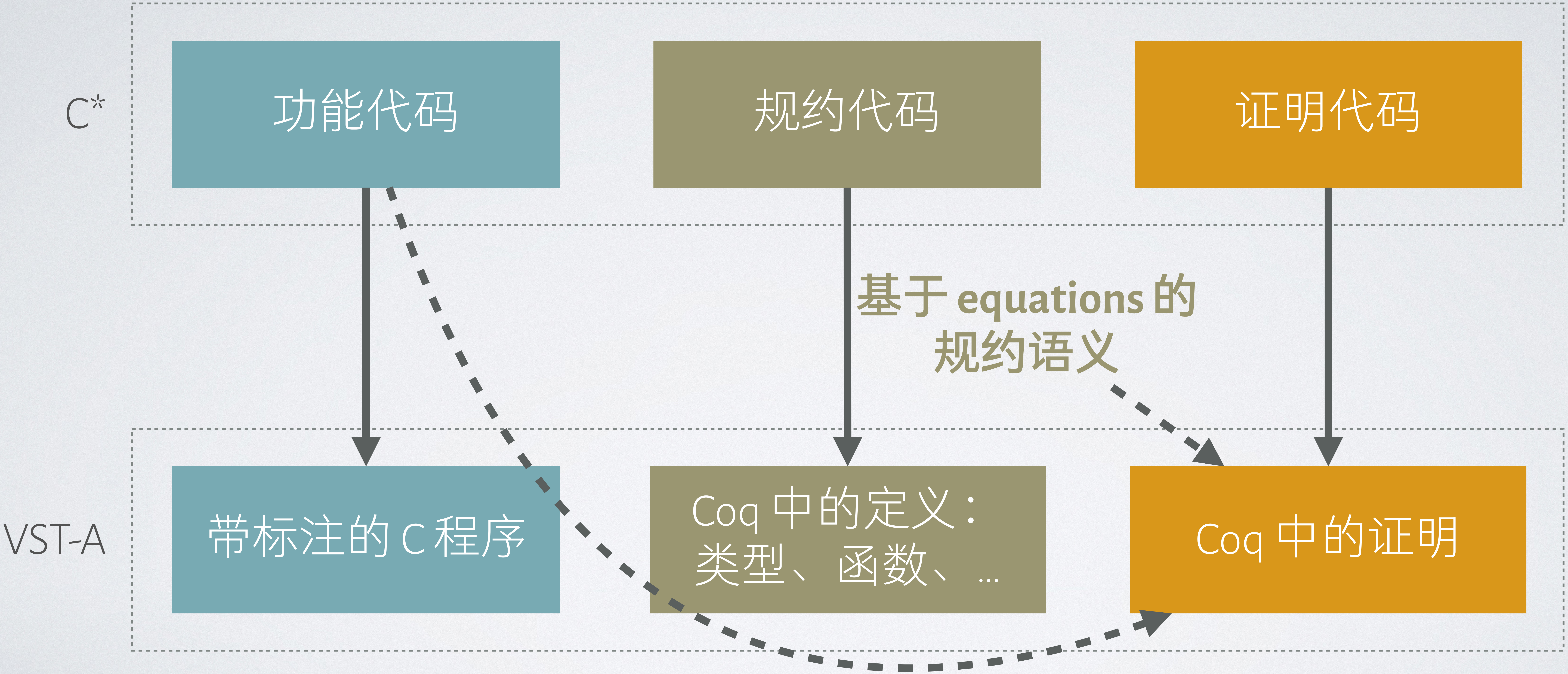
短期原型实现方案：基于 VST-A



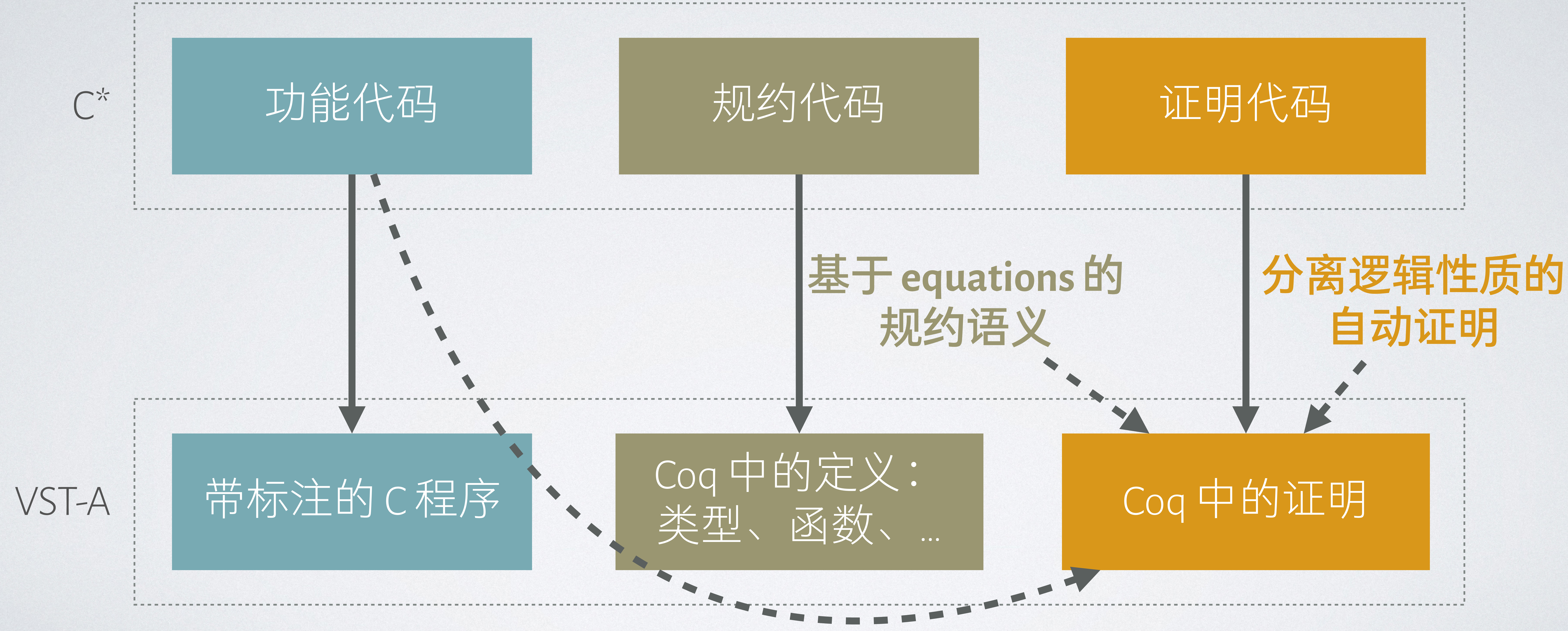
短期原型实现方案：基于 VST-A



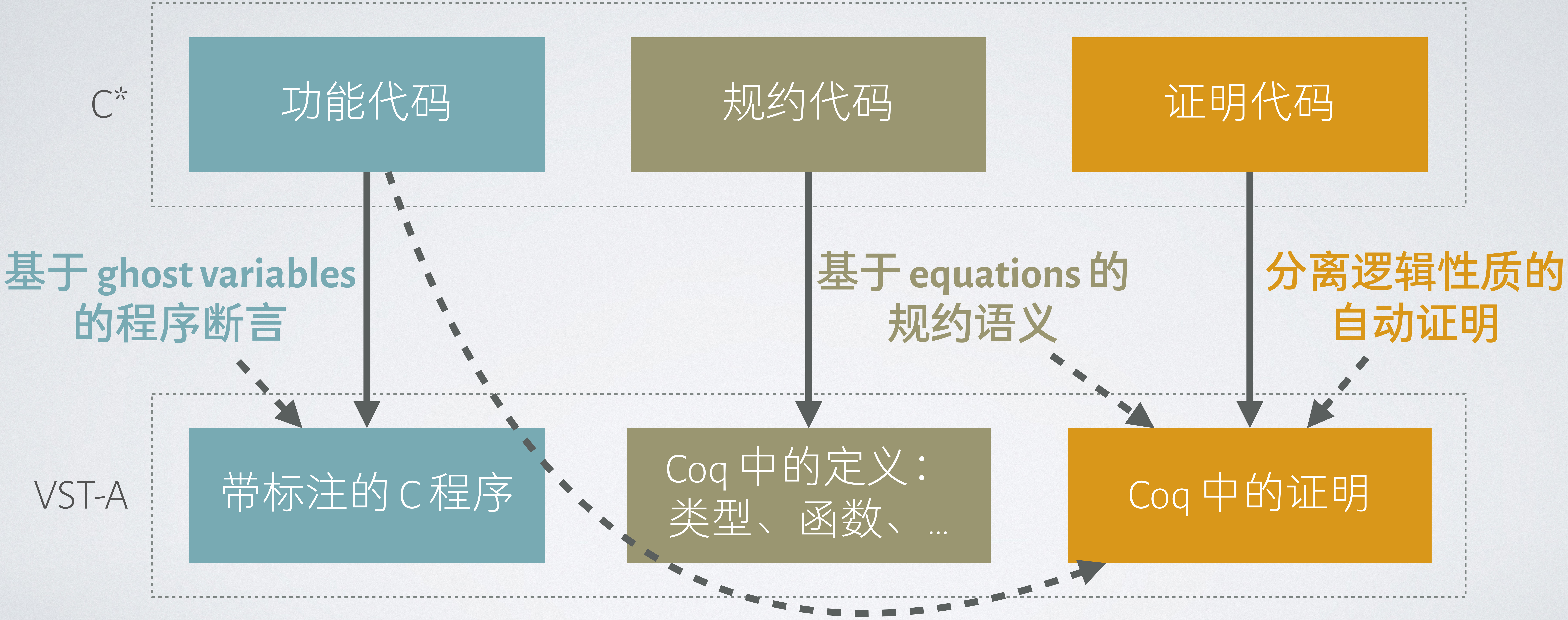
短期原型实现方案：基于 VST-A



短期原型实现方案：基于 VST-A



短期原型实现方案：基于 VST-A



理想的工作流

理想的工作流

- 所有的操作在**统一的环境（IDE）**中进行

理想的工作流

- 所有的操作在**统一的环境（IDE）**中进行
- 编写**功能代码**时，IDE 根据用户标注的前后条件、断言等提示需要证明的定理
    - VST-A 的符号执行

理想的工作流

- 所有的操作在**统一的环境（IDE）**中进行
- 编写**功能代码**时，IDE 根据用户标注的前后条件、断言等提示需要证明的定理
  - VST-A 的符号执行
- 编写**规约代码**时，IDE 自动维护证明中可以使用的定理
  - 数据结构：归纳法
  - 规约函数：到基于等式的表示法的双向变换

理想的工作流

- 所有的操作在**统一的环境（IDE）**中进行
- 编写**功能代码**时，IDE 根据用户标注的前后条件、断言等提示需要证明的定理
  - VST-A 的符号执行
- 编写**规约代码**时，IDE 自动维护证明中可以使用的定理
  - 数据结构：归纳法
  - 规约函数：到基于等式的表示法的双向变换
- 编写**证明代码**时，IDE 提示当前的证明状态
  - 与 Coq 不同，CStar 的证明是需要具体运行的
  - 可以使用部分求值等技术

C*：面向形式化证明的程序设计语言

功能代码

普通 C 代码
支持符号执行

规约代码

前、后条件
循环不变式
断言、规约函数
分离逻辑

证明代码

证明规约之间的关系
命令式风格的证明

命令式风格

代码对程序状态进行操作
规约对程序状态进行描述
证明对证明状态进行操作