



# 开发与证明一体化的编程范式

王迪



# 如何开发高安全级别的软件？



# 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

# 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



# 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**



# 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





从语言/工具的层面，我们能为软件的形式化证明提供什么？



# 编程语言能提供什么？



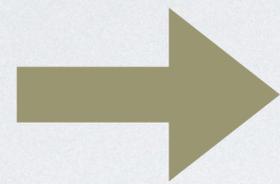
# 编程语言能提供什么？

## B

- ◎ 1969 年
- ◎ **无类型系统**
- ◎ 性能差，不安全

# 编程语言能提供什么？

## B



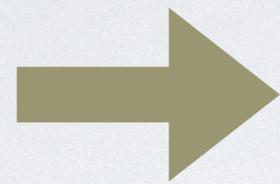
## C

- ◎ 1969 年
- ◎ **无类型系统**
- ◎ 性能差，不安全

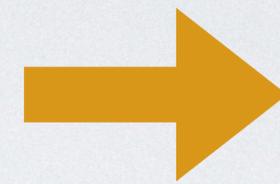
- ◎ 1972 年
- ◎ **简单类型系统**
- ◎ 性能好，基本安全

# 编程语言能提供什么？

B



C



R<sub>ust</sub>

- ◎ 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

# 编程语言能提供什么？



- ◎ 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

# 编程语言能提供什么？



- ◎ 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

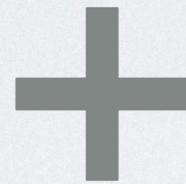


如何在写 C 程序的时候就尽量保证程序安全且正确？

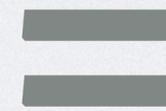


如何在写 C 程序的时候就尽量保证程序安全且正确？

C 程序



各种编程范式  
(比如防御式编程)



安全的 C 程序

如何在写 C 程序的时候就尽量保证程序安全且正确？



参考 Rust，是否可以把一类编程范式**内置**到编程语言里？



# 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

```
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?
- 如何设计编程语言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?
- 如何设计编程语言来提升代码的可信度?
- 通过 **Contracts** 为代码添加信息



# 编程语言能提供什么？

```
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;
}
```

# 编程语言能提供什么？

```
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**：设计编程语言，把动态检查 contracts 的过程静态化



# 编程语言能提供什么？

```
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**：设计编程语言，把动态检查 contracts 的过程静态化
- ◎ **No Free Lunch**：程序员需要在编程时标注前后条件、循环不变式以及断言

# 编程语言能提供什么？

```
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**：设计编程语言，把动态检查 contracts 的过程静态化
- ◎ **No Free Lunch**：程序员需要在编程时标注前后条件、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

# 编程语言能提供什么？

```
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;
}
```

静态分析循环体  
达成的效果



- ◎ **核心**：设计编程语言，把动态检查 contracts 的过程静态化
- ◎ **No Free Lunch**：程序员需要在编程时标注前后条件、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



# 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;  
}
```

$r=r\theta$ ,  $b=b\theta$ ,  $e=e\theta$

# 静态分析代码的效果

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

$r=r_0, b=b_0, e=e_0$



$r = (e_0 \% 2 == 1) ? b_0 * r_0 : r_0$   
 $b = b_0 * b_0$   
 $e = e_0 / 2$

# 静态分析代码的效果

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

$r=r_0, b=b_0, e=e_0$



可以通过符号执行自动实现

$r = (e_0 \% 2 == 1) ? b_0 * r_0 : r_0$

$b = b_0 * b_0$

$e = e_0 / 2$



# 证明性质之间的关系

$r=r\theta, b=b\theta, e=e\theta$

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

$r = (e\theta \% 2 == 1) ? b\theta * r\theta : r\theta$   
 $b = b\theta * b\theta$   
 $e = e\theta / 2$



# 证明性质之间的关系

$$r=r\theta, b=b\theta, e=e\theta$$

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

$$\begin{aligned} & POW(b\theta, e\theta) * r\theta == POW(x, y) \\ \implies & POW(b, e) * r == POW(x, y) \end{aligned}$$

$$\begin{aligned} r &= (e\theta \% 2 == 1) ? b\theta * r\theta : r\theta \\ b &= b\theta * b\theta \\ e &= e\theta / 2 \end{aligned}$$

# 证明性质之间的关系

$r=r\theta, b=b\theta, e=e\theta$

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

○ 采取交互式定理证明

$POW(b\theta, e\theta) * r\theta == POW(x, y)$   
 $\implies POW(b, e) * r == POW(x, y)$

$r = (e\theta \% 2 == 1) ? b\theta * r\theta : r\theta$   
 $b = b\theta * b\theta$   
 $e = e\theta / 2$

# 证明性质之间的关系

$r=r\theta, b=b\theta, e=e\theta$

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

- 采取交互式定理证明
- 采取基于 SMT 等的自动证明技术

$POW(b\theta, e\theta) * r\theta == POW(x, y)$   
 $\implies POW(b, e) * r == POW(x, y)$

$r = (e\theta \% 2 == 1) ? b\theta * r\theta : r\theta$   
 $b = b\theta * b\theta$   
 $e = e\theta / 2$

# 证明性质之间的关系

$r=r\theta, b=b\theta, e=e\theta$

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

- 采取交互式定理证明
- 采取基于 SMT 等的自动证明技术
- 采取 .....

$$\begin{aligned} & \text{POW}(b\theta, e\theta) * r\theta == \text{POW}(x, y) \\ \implies & \text{POW}(b, e) * r == \text{POW}(x, y) \end{aligned}$$

$$\begin{aligned} r &= (e\theta \% 2 == 1) ? b\theta * r\theta : r\theta \\ b &= b\theta * b\theta \\ e &= e\theta / 2 \end{aligned}$$



# 开发与证明一体化的编程范式：以 C 语言为例

功能代码

规约代码

证明代码



# 开发与证明一体化的编程范式：以 C 语言为例

功能代码

普通 C 代码

规约代码

证明代码



# 开发与证明一体化的编程范式：以 C 语言为例

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



# 开发与证明一体化的编程范式：以 C 语言为例

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



# 开发与证明一体化的编程范式：以 C 语言为例

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

需要统一为 C 风格



# 开发与证明一体化的编程范式：以 C 语言为例

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

**需要统一为 C 风格**

如何支持 C 风格的规约？



# 开发与证明一体化的编程范式：以 C 语言为例

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

**需要统一为 C 风格**

如何支持 C 风格的规约？

如何支持 C 风格的证明？



我们先来看证明的部分



# 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?

一个简单的  
规约函数

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



# 如何写证明?

一个简单的  
规约函数

```
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

一个简单的  
规约函数

```
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 I1 = refl(`POW(x,0)`);
    // I1: |- 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 11 = refl(`POW(x,0)`);
  // 11: |- POW(x,0)==POW(x,0);
  thm 12 = rewrite(11, `POW(x,0)`);
  // 12: |- 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 C 风格来操作证明对象



# 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 C 风格来操作证明对象
- 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 C 风格来操作证明对象
- 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 11 = smt(`POW(x,0) == 1`);  
// 11: |- 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 C 风格来操作证明对象
- 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 11 = smt(`POW(x,0) == 1`);  
// 11: |- 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？



# C 风格的后向证明



# C 风格的后向证明

- **后向证明：**从目标出发，通过 tactic 变换目标



# C 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  $A \wedge B$ ，通过 split tactic 可以得到两个子目标 A 和 B



# C 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  $A \wedge B$ ，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！



# C 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  $A \wedge B$ , 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
```



# C 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  $A \wedge B$ , 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)
```



# C 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  $A \wedge B$ ，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
```



# C 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



# C 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
```



# C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)
```



# C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)
```



# C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`);
```



# C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)
```



# C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);
```



# C 风格的等式证明



# C 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



# C 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



# C 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`);
```



# C 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
```



# C 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
```



# C 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);
```



我们再来看规约的部分



# 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;
}
```

- 「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);
}
```

# 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);
}
```

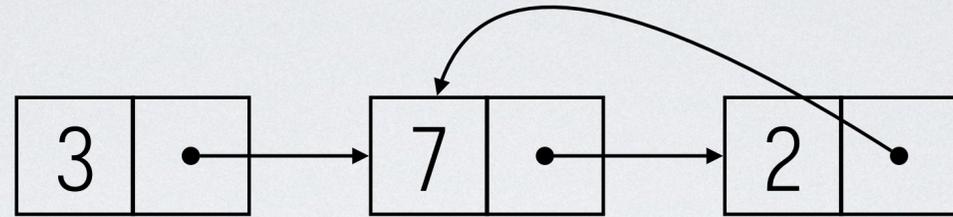
```
bool le_seg(int x, int[] A, int lo, int hi)
//@require ...
{
    if (lo == hi) return true;
    return x <= A[lo] && le_seg(x, A, lo + 1, hi);
}
```

```
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);
}
```

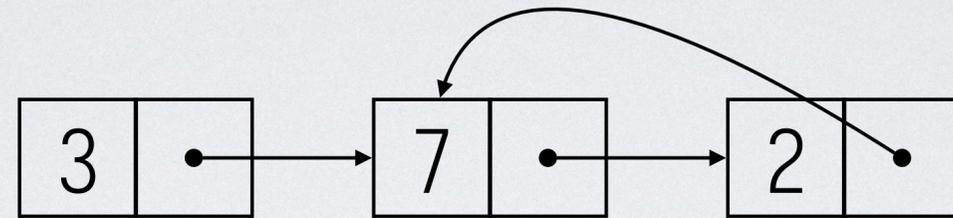


# 分离逻辑：适用于描述堆相关的规约

# 分离逻辑：适用于描述堆相关的规约

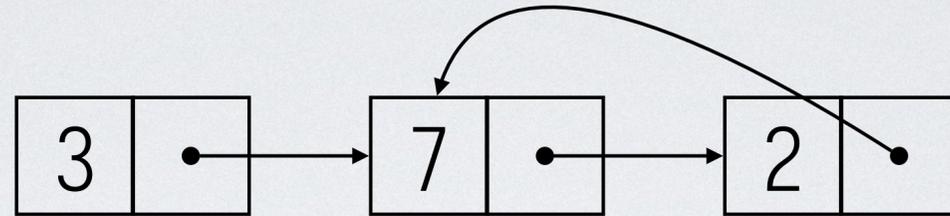


# 分离逻辑：适用于描述堆相关的规约



$11 \sim > (3, 12) * 12 \sim > (7, 13) * 13 \sim > (2, 12)$

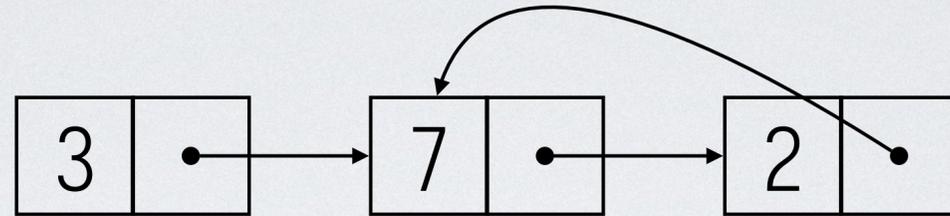
# 分离逻辑：适用于描述堆相关的规约



$11 \sim > (3, 12) * 12 \sim > (7, 13) * 13 \sim > (2, 12)$

$loc \sim > data$  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：适用于描述堆相关的规约

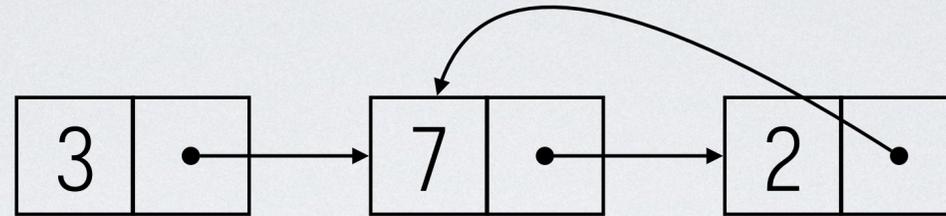


$11 \sim > (3, 12) * 12 \sim > (7, 13) * 13 \sim > (2, 12)$

$loc \sim > data$  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：适用于描述堆相关的规约



$11 \sim > (3, 12) * 12 \sim > (7, 13) * 13 \sim > (2, 12)$

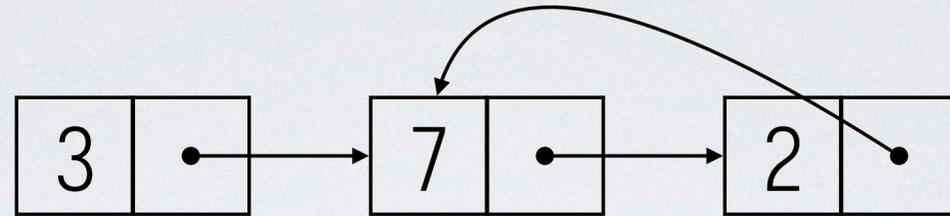
$loc \sim > data$  表示一块堆空间，其中只有一个地址  $loc$ ，指向的数据为  $data$

$*$  表示合并两块地址**不相交**的堆空间

```

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：适用于描述堆相关的规约



$11 \sim > (3, 12) * 12 \sim > (7, 13) * 13 \sim > (2, 12)$

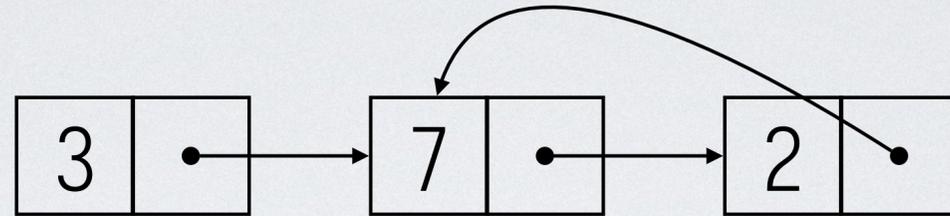
$loc \sim > data$  表示一块堆空间，其中只有一个地址  $loc$ ，指向的数据为  $data$

$*$  表示合并两块地址**不相交**的堆空间

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：适用于描述堆相关的规约



$11 \sim > (3, 12) * 12 \sim > (7, 13) * 13 \sim > (2, 12)$

$loc \sim > data$  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

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);
}

```

$start$  到  $end$  描述了一个单链表，其内容为  $A[lo..hi)$



# 分离逻辑：解耦内存布局和计算性质

```
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：解耦内存布局和计算性质

```
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));
{
  ...
}
```

解耦内存布局  
和计算性质



# 分离逻辑：解耦内存布局和计算性质

```
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));
{
  ...
}
```

解耦内存布局  
和计算性质

```
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：解耦内存布局和计算性质

```
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));
{
  ...
}
```

解耦内存布局  
和计算性质

```
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;
}
```

规约函数可使用 C 风格循环



# 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;
}
```

$$y==0 \implies \text{POW}(x, y) == 1$$

$$y>0 \implies \text{POW}(x, y) == \text{POW}(x, y-1) * x$$



# 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
```

可以利用关于规约函数满足的等式来进行性质的证明

```
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;
}
```

$y==0 \implies \text{POW}(x, y) == 1$

$y>0 \implies \text{POW}(x, y) == \text{POW}(x, y-1) * x$

# 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;
}
```



# 开发与证明一体化的编程范式：以 C 语言为例

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  
**C 风格的证明**



# 开发与证明一体化的编程范式：以 C 语言为例

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  
C 风格的证明

如何抽取出更一般的设计范式？



# 开发与证明一体化的编程范式

功能代码

规约代码

证明代码

# 开发与证明一体化的编程范式



# 开发与证明一体化的编程范式

功能代码

规约代码

证明代码

**程序逻辑**

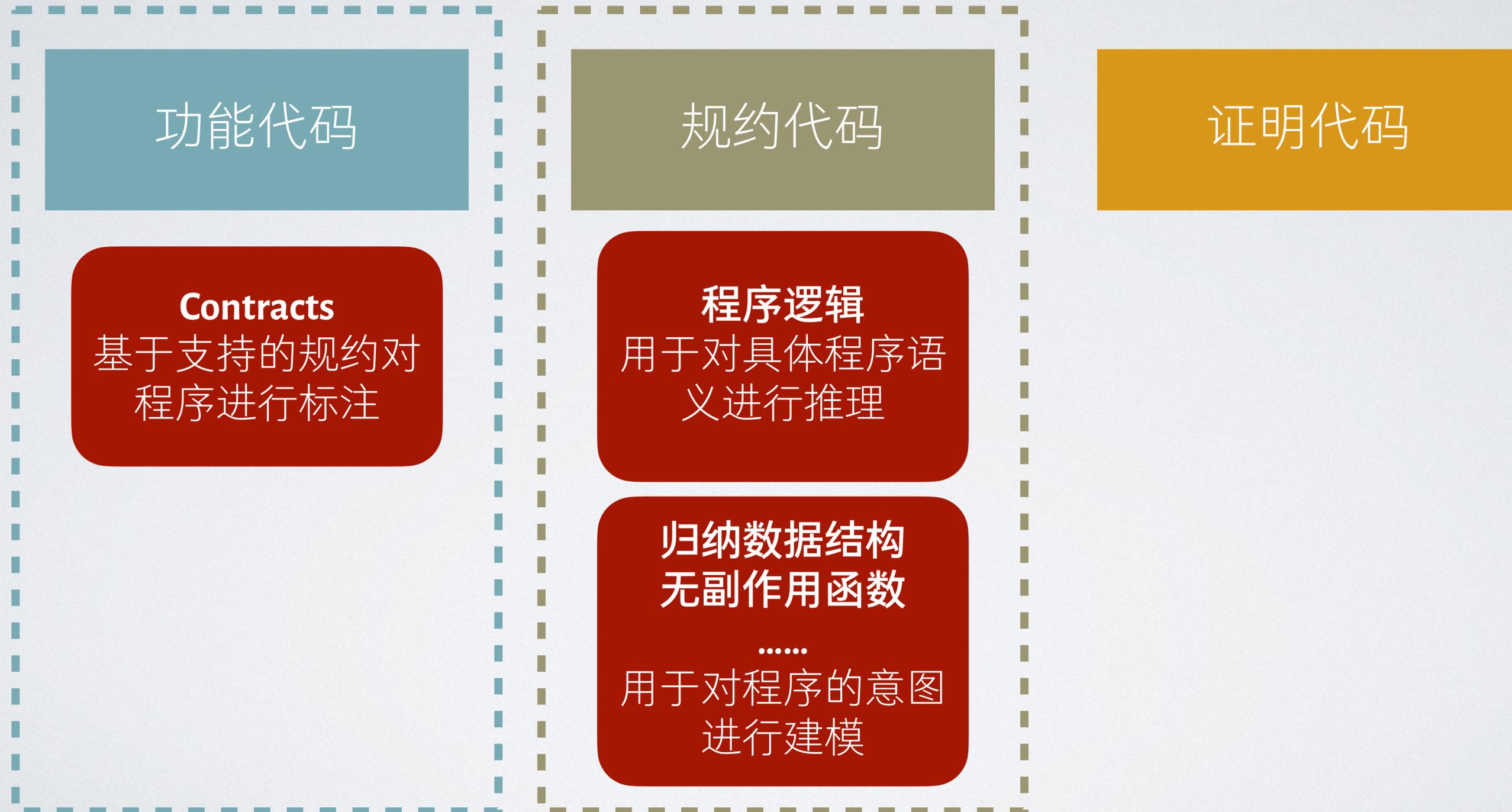
用于对具体程序语义进行推理

**归纳数据结构  
无副作用函数**

.....

用于对程序的意图进行建模

# 开发与证明一体化的编程范式





# 开发与证明一体化的编程范式

功能代码

**Contracts**

基于支持的规约对  
程序进行标注

**符号执行**

基于程序逻辑进行  
最强后条件推理并  
生成验证条件

规约代码

**程序逻辑**

用于对具体程序语  
义进行推理

**归纳数据结构  
无副作用函数**

.....

用于对程序的意图  
进行建模

证明代码

# 开发与证明一体化的编程范式

## 功能代码

### Contracts

基于支持的规约对程序进行标注

### 符号执行

基于程序逻辑进行最强后条件推理并生成验证条件

## 规约代码

### 程序逻辑

用于对具体程序语义进行推理

### 归纳数据结构 无副作用函数

.....  
用于对程序的意图进行建模

## 证明代码

### LCF证明核心

以X语言本身为元语言并支持定理、目标等抽象结构

# 开发与证明一体化的编程范式

## 功能代码

### Contracts

基于支持的规约对程序进行标注

### 符号执行

基于程序逻辑进行最强后条件推理并生成验证条件

## 规约代码

### 程序逻辑

用于对具体程序语义进行推理

### 归纳数据结构 无副作用函数

.....  
用于对程序的意图进行建模

## 证明代码

### LCF证明核心

以X语言本身为元语言并支持定理、目标等抽象结构

### 外部证明能力

暴露定理、目标的接口以方便用户翻译到其它证明工具



# 开发与证明一体化的编程范式：以 **X** 语言为例



# 开发与证明一体化的编程范式：以 **X** 语言为例

X 语言



# 开发与证明一体化的编程范式：以 **X** 语言为例

X 语言

函数式/声明式  
规约语言  
(可为 X 子集)



# 开发与证明一体化的编程范式：以 **X** 语言为例

X 语言

函数式/声明式  
规约语言  
(可为 X 子集)

Contract 标注



# 开发与证明一体化的编程范式：以 **X** 语言为例

X 语言

函数式/声明式  
规约语言  
(可为 X 子集)

Contract 标注

语言本体

# 开发与证明一体化的编程范式：以 **X** 语言为例

X 语言

函数式/声明式  
规约语言  
(可为 X 子集)

Contract 标注

语言本体

库/包/...

# 开发与证明一体化的编程范式：以 **X** 语言为例

X 语言

函数式/声明式  
规约语言  
(可为 X 子集)

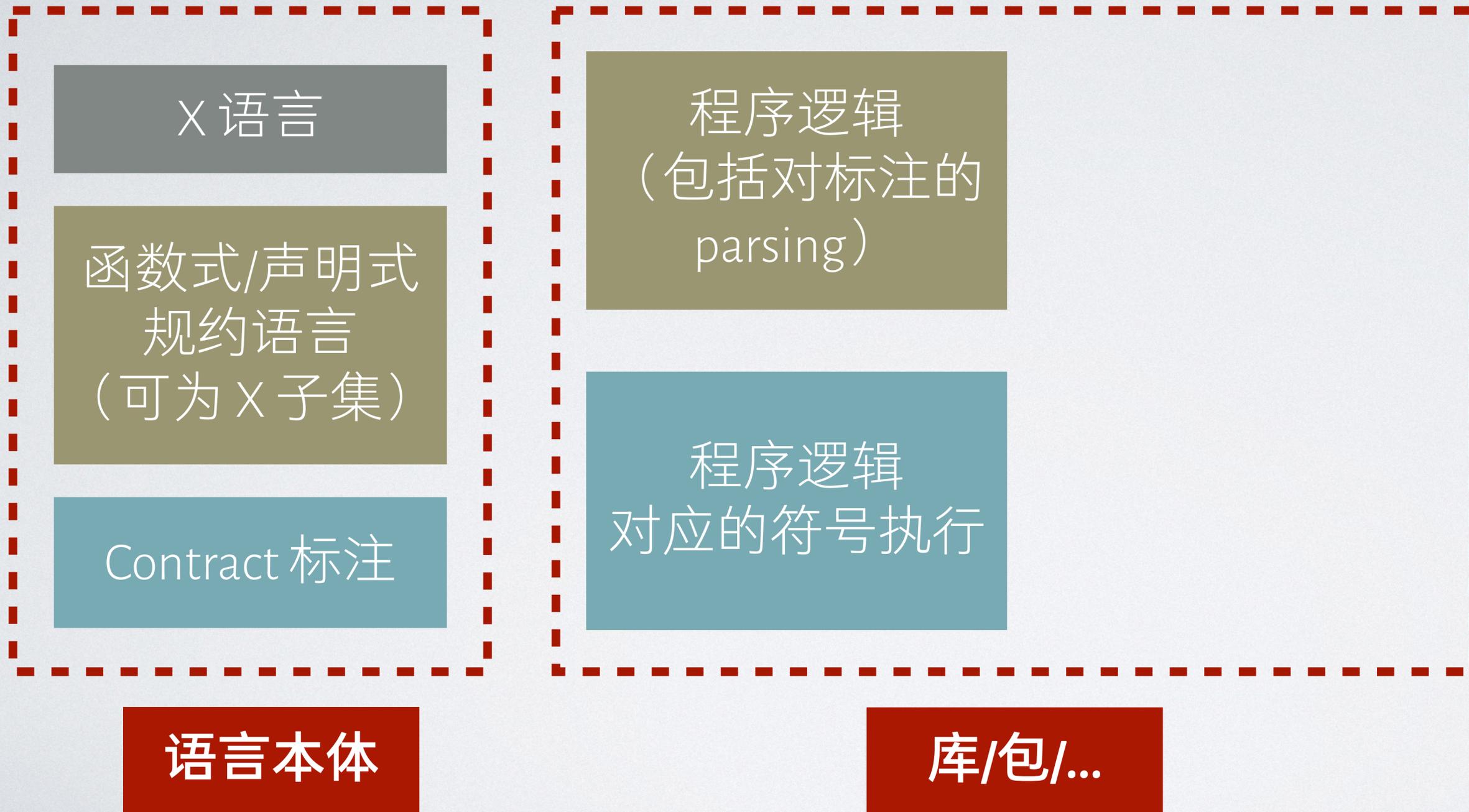
Contract 标注

语言本体

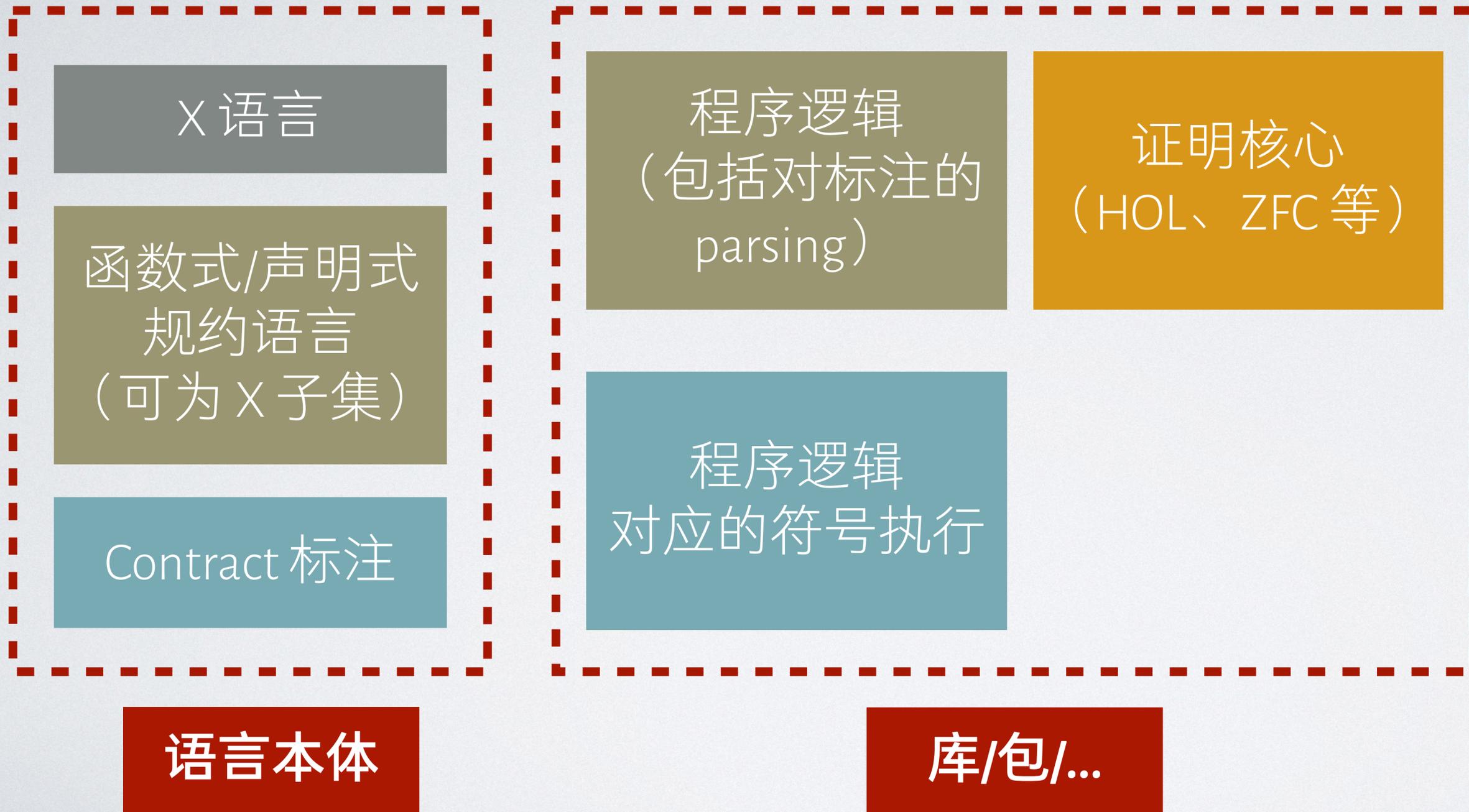
程序逻辑  
(包括对标注的  
parsing)

库/包/...

# 开发与证明一体化的编程范式：以 **X** 语言为例

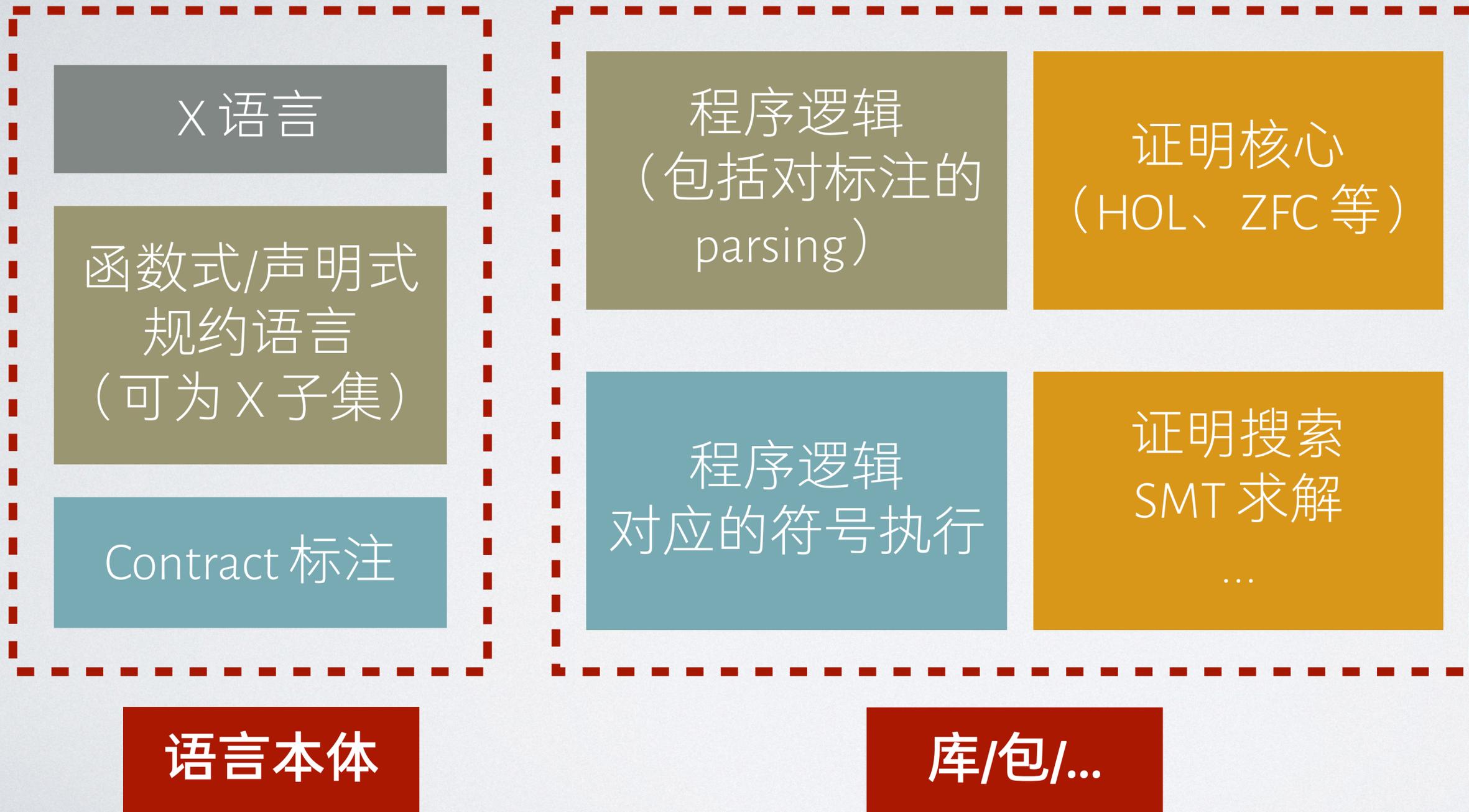


# 开发与证明一体化的编程范式：以 **X** 语言为例





# 开发与证明一体化的编程范式：以 X 语言为例



# 开发与证明一体化的编程范式：以 X 语言为例



# 开发与证明一体化的编程范式：以 X 语言为例



# 开发与证明一体化的编程范式：以 X 语言为例

