

优化与目标代码生成

杨策

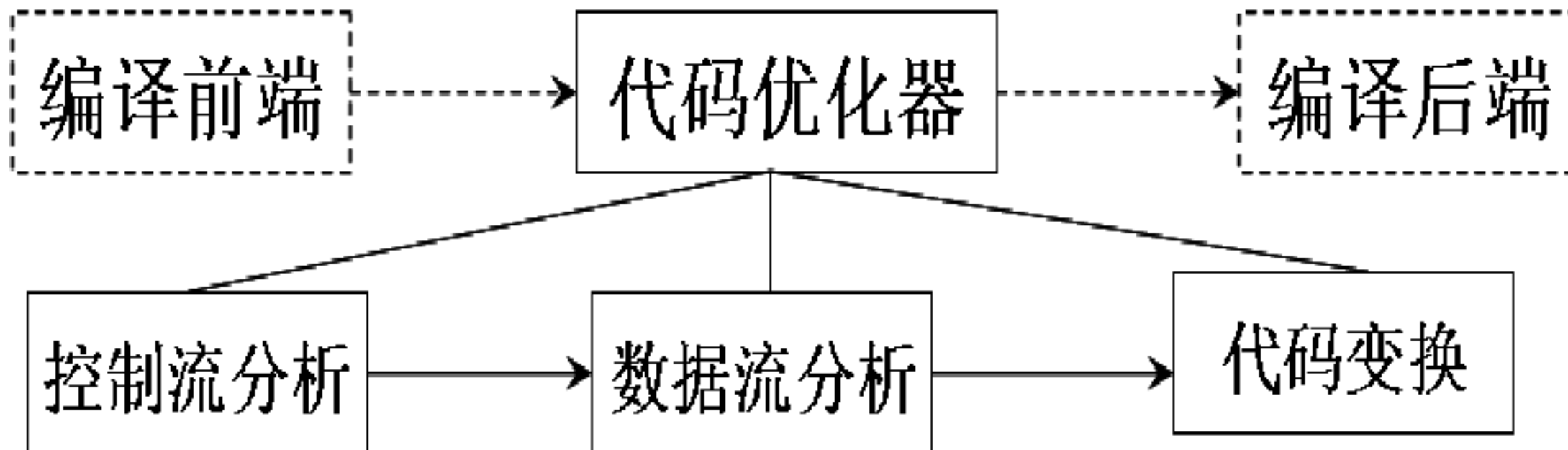
2023-05-06

目录

1. 优化概述
2. 局部优化
3. 全局优化
4. 目标代码生成-llvm

1.1 优化的概念

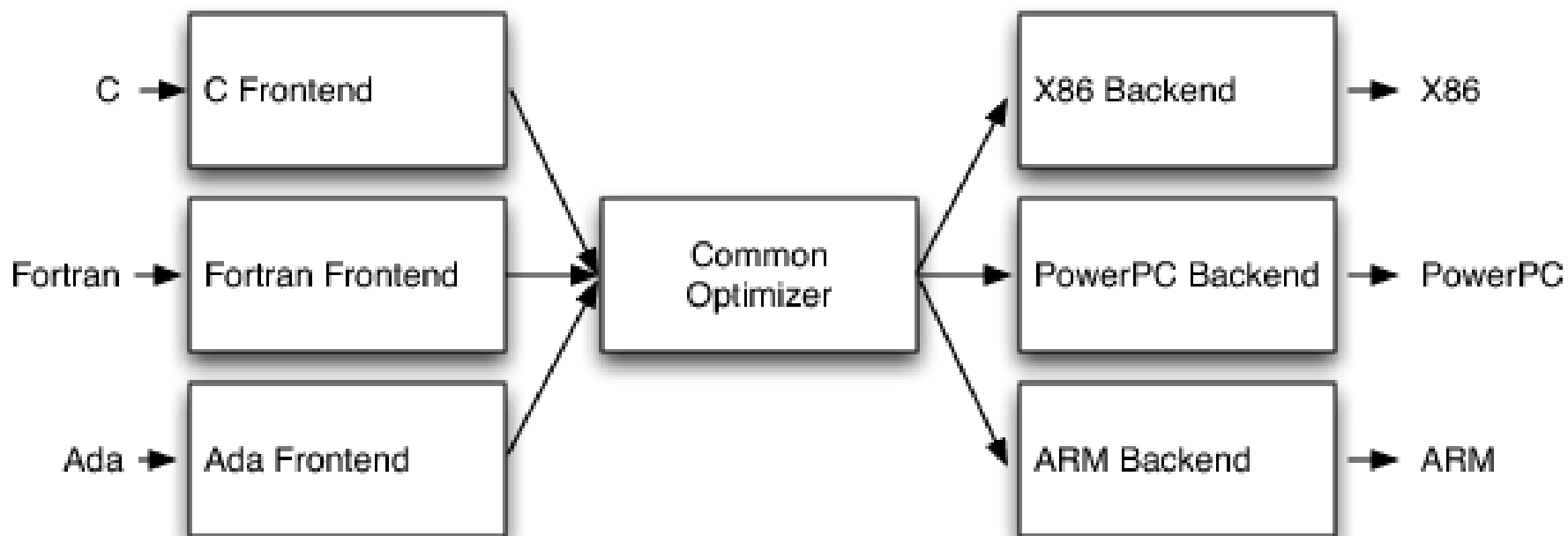
- 优化与其他组件的关系



代码优化器的地位和结构

1.1 优化的概念

- llvm 架构

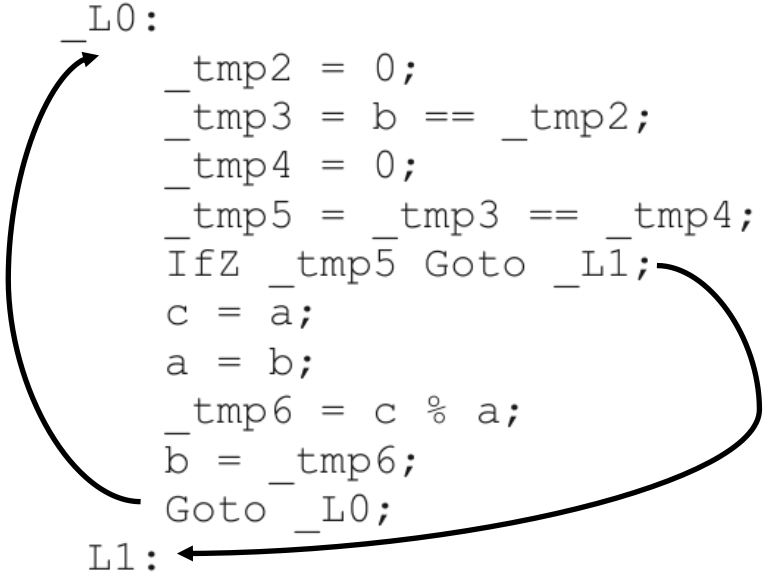


1.1 优化的概念

- 等价原则
 - 经过优化后不应改变程序运行的结果
 - 语义保留：冒泡排序->快速排序
- 有效原则
 - 使优化后所产生的目标代码运行时间较短，占用的存储空间较小
- 合算原则
 - 尽可能以较低的代价取得较好的优化效果

1.2 基本块与控制流图

```
main:
    BeginFunc 40;
    _tmp0 = LCall _ReadInteger;
    a = _tmp0;
    _tmp1 = LCall _ReadInteger;
    b = _tmp1;
_L0:
    _tmp2 = 0;
    _tmp3 = b == _tmp2;
    _tmp4 = 0;
    _tmp5 = _tmp3 == _tmp4;
    IfZ _tmp5 Goto _L1;
    c = a;
    a = b;
    _tmp6 = c % a;
    b = _tmp6;
    Goto _L0;
_L1:
    PushParam a;
    LCall _PrintInt;
    PopParams 4;
    EndFunc;
```



The diagram illustrates the control flow of the provided code. It shows two basic blocks: `_L0` and `_L1`. `_L0` contains a loop that branches to `_L1` if the zero flag is set (IfZ). `_L1` contains the code to print the value of `a` and then returns. A curved arrow points from the `Goto _L0` statement back to the `_L0` label, indicating a loop.

- 基本块
 - 代码顺序执行
- 控制流图 (CFG)
 - 基本块组成的图
 - 基本块是结点
 - 跳转关系是边

1.2 基本块与控制流图

main:

```
BeginFunc 40;  
_tmp0 = LCall _ReadInteger;  
a = _tmp0;  
_tmp1 = LCall _ReadInteger;  
b = _tmp1;
```

_L0:

```
_tmp2 = 0;  
_tmp3 = b == _tmp2;  
_tmp4 = 0;  
_tmp5 = _tmp3 == _tmp4;  
IfZ _tmp5 Goto _L1;  
c = a;  
a = b;  
_tmp6 = c % a;  
b = _tmp6;  
Goto _L0;
```

_L1:

```
PushParam a;  
LCall _PrintInt;  
PopParams 4;  
EndFunc;
```

```
_tmp0 = LCall _ReadInteger;  
a = _tmp0 ;  
_tmp1 = LCall _ReadInteger;  
b = _tmp1 ;
```

```
_tmp2 = 0 ;  
_tmp3 = b == _tmp2 ;  
_tmp4 = 0 ;  
_tmp5 = _tmp3 == _tmp4 ;  
IfZ _tmp5 Goto _L1 ;
```

```
c = a ;  
a = b ;  
_tmp6 = c % a ;  
b = _tmp6 ;  
Goto _L0 ;
```

```
PushParam a ;  
LCall _PrintInt ;  
PopParams 4 ;
```

1.3 优化分类

- 局部优化
 - 单个基本块内部优化
- 全局优化
 - 一个函数内所有基本块联合优化
- 跨函数优化
 - 多个函数的代码联合优化

2.1 算术化简 (Algebraic Simplification)

- 删除某些表达式

- $x = x + 0$

- $x = x * 1$

- 化简某些表达式

$$x := x * 0 \quad \Rightarrow \quad x := 0$$

$$y := y ** 2 \quad \Rightarrow \quad y := y * y$$

$$x := x * 8 \quad \Rightarrow \quad x := x \ll 3$$

$$x := x * 15 \quad \Rightarrow \quad t := x \ll 4; x := t - x$$

2.2 常量合并 (Constant Folding)

- 编译期计算
 - $x=2+2 \Rightarrow x=4$
 - 字符串连接
 - 问号表达式
- C++
 - 模板元编程
 - `constexpr`

2.2 常量合并

- 编译期计算
 - $x=2+2 \Rightarrow x=4$
 - 字符串连接
 - 问号表达式
- C++
 - 模板元编程
 - constexpr

```
template <size_t N>
struct factorial
{
    enum : size_t { value = N * factorial<N - 1>::value };
};

template <>
struct factorial<0>
{
    enum : size_t { value = 1 };
};

int main(void)
{
    std::cout << factorial<10>::value << std::endl;
    return 0;
}
```

2.2 常量合并

- 编译期计算
 - $x=2+2 \Rightarrow x=4$
 - 字符串连接
 - 问号表达式
- C++
 - 模板元编程
 - **constexpr**

```
#include <iostream>

constexpr double sum(double d)
{
    return d > 0 ? d + sum(d - 1) : 0;
}

int main(void)
{
    constexpr double d = sum(5);
    std::cout << d;
}
```

指示性，非强制性
类似于inline

2.3 控制流优化

- 消除不可达基本块
 - 从起点出发不可达
 - 图算法，类似于垃圾回收
- 减少空间代码
- 可能让程序运行速度更快
 - 缓存命中

2.4 静态单赋值 (SSA)

- Static Single Assignment (SSA)

- 每个变量/寄存器只赋值一次
- 有的变量需要重命名
- 方便后续优化

$x := z + y$

$a := x$

$x := 2 * x$

\Rightarrow

$b := z + y$

$a := b$

$x := 2 * b$

2.5 删除公共子表达式

- 基于SSA
- 赋值语句右部相同

$x := y + z$

...

$w := y + z$

\Rightarrow

$x := y + z$

...

$w := x$

2.6 复写传播

- 基于SSA
- 变量复制，左边用右边替换

```
b := z + y  
a := b  
x := 2 * a
```

\Rightarrow

```
b := z + y  
a := b  
x := 2 * b
```


2.6 复写传播&常量合并

- 结合之后化简

```
a := 5  
x := 2 * a  
y := x + 6  
t := x * y
```

⇒

```
a := 5  
x := 10  
y := 16  
t := 160
```

2.6 复写传播&删除无用代码

- 死代码消除
 - 仅赋值，未使用

$b := z + y$		$b := z + y$		$b := z + y$
$a := b$	\Rightarrow	$a := b$	\Rightarrow	$x := 2 * b$
$x := 2 * a$		$x := 2 * b$		

假设变量a没有在其他地方使用

2.7 局部优化

- 原则

- 小步
- 多趟
- 多种方法共同作用
- 无法优化时停止
- 或者其他时候停止

优化前代码

```
a = x**2
b = 3
c = x
d = c*c
e = b*2
f = a+d
g = e*f
```

2.7 局部优化

- 原则

- 小步
- 多趟
- 多种方法共同作用
- 无法优化时停止
- 或者其他时候停止

优化前代码

$a = x^{**}2$

$b = 3$

$c = x$

$d = c * c$

$e = b * 2$

$f = a + d$

$g = e * f$

算术优化

$a = x * x$

$b = 3$

$c = x$

$d = c * c$

$e = b << 1$

$f = a + d$

$g = e * f$

2.7 局部优化

- 原则

- 小步
- 多趟
- 多种方法共同作用
- 无法优化时停止
- 或者其他时候停止

优化前代码

a = x*x

b = 3

c = x

d = c*c

e = b<<1

f = a+d

g = e*f

复写传播

a = x*x

b = 3

c = x

d = x*x

e = 3<<1

f = a+d

g = e*f

2.7 局部优化

- 原则

- 小步
- 多趟
- 多种方法共同作用
- 无法优化时停止
- 或者其他时候停止

优化前代码

```
a = x*x
b = 3
c = x
d = x*x
e = 3<<1
f = a+d
g = e*f
```

常数合并

```
a = x*x
b = 3
c = x
d = x*x
e = 6
f = a+d
g = e*f
```

2.7 局部优化

- 原则

- 小步
- 多趟
- 多种方法共同作用
- 无法优化时停止
- 或者其他时候停止

优化前代码

$a = x * x$

$b = 3$

$c = x$

$d = x * x$

$e = 6$

$f = a + d$

$g = e * f$

删除公共子表达式

$a = x * x$

$b = 3$

$c = x$

$d = a$

$e = 6$

$f = a + d$

$g = e * f$

2.7 局部优化

- 原则

- 小步
- 多趟
- 多种方法共同作用
- 无法优化时停止
- 或者其他时候停止

优化前代码

a = x*x

b = 3

c = x

d = a

e = 6

f = a+d

g = e*f

复写传播

a = x*x

b = 3

c = x

d = a

e = 6

f = a+a

g = 6*f

2.7 局部优化

- 原则

- 小步
- 多趟
- 多种方法共同作用
- 无法优化时停止
- 或者其他时候停止

优化前代码

a = x*x

b = 3

c = x

d = a

e = 6

f = a+a

g = 6*f

删除无用代码

a = x*x

f = a+a

g = 6*f

优化完成!

2.8 窥孔优化 (Peephole Optimizations)

- 用于中间代码
 - 前述优化技术
- 窥孔优化
 - 用于目标代码
 - 与中间代码优化类似
 - 与指令有关

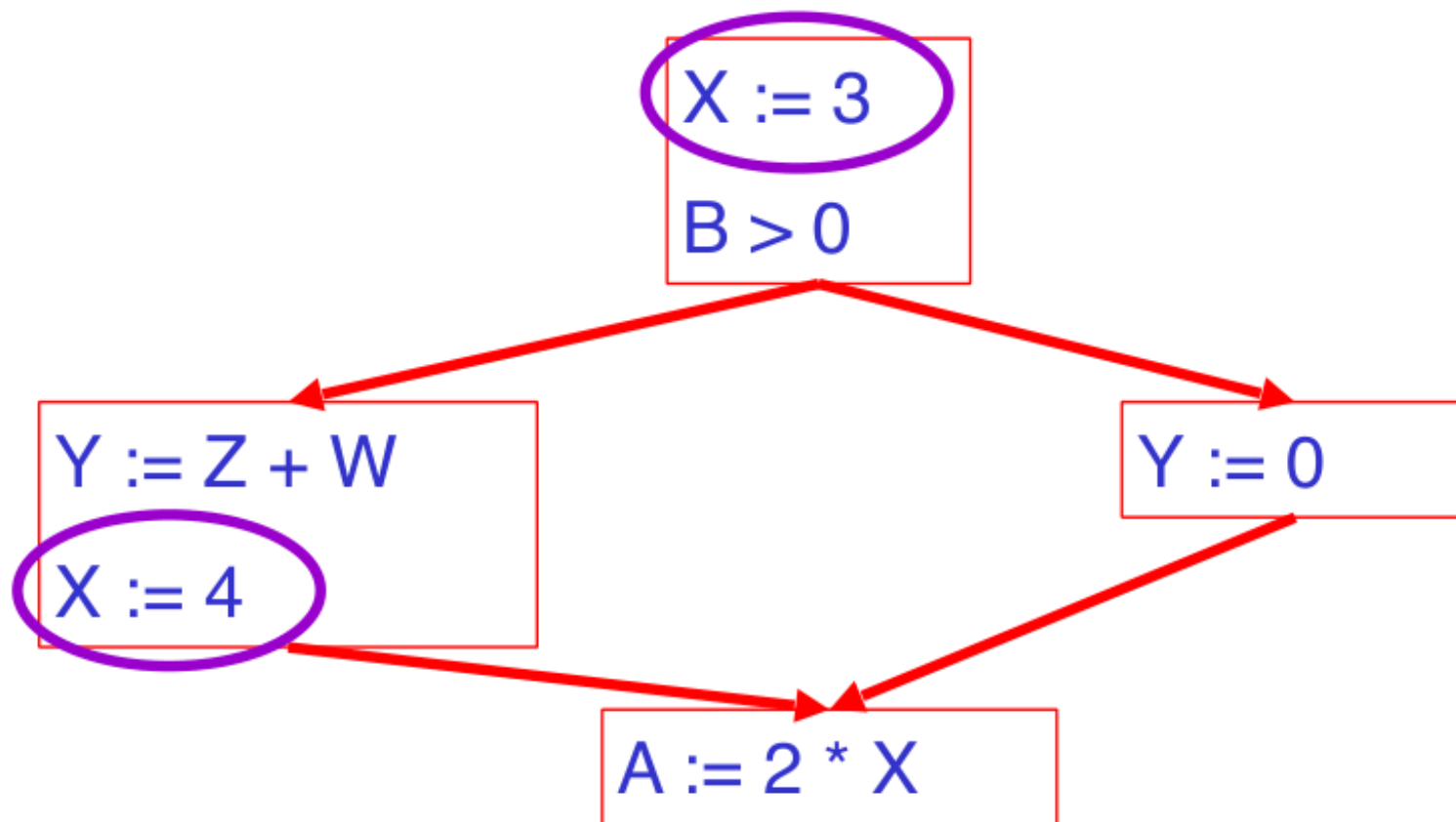


3.1 循环优化

- 代码外提
- 强度削弱
- 删除归纳变量

3.2 全局常量传播

- 需要确认变量取值
 - 是否是常量
 - 具体值是多少
- 从前向后推导
 - 多条简单规则
 - 一步一步更新

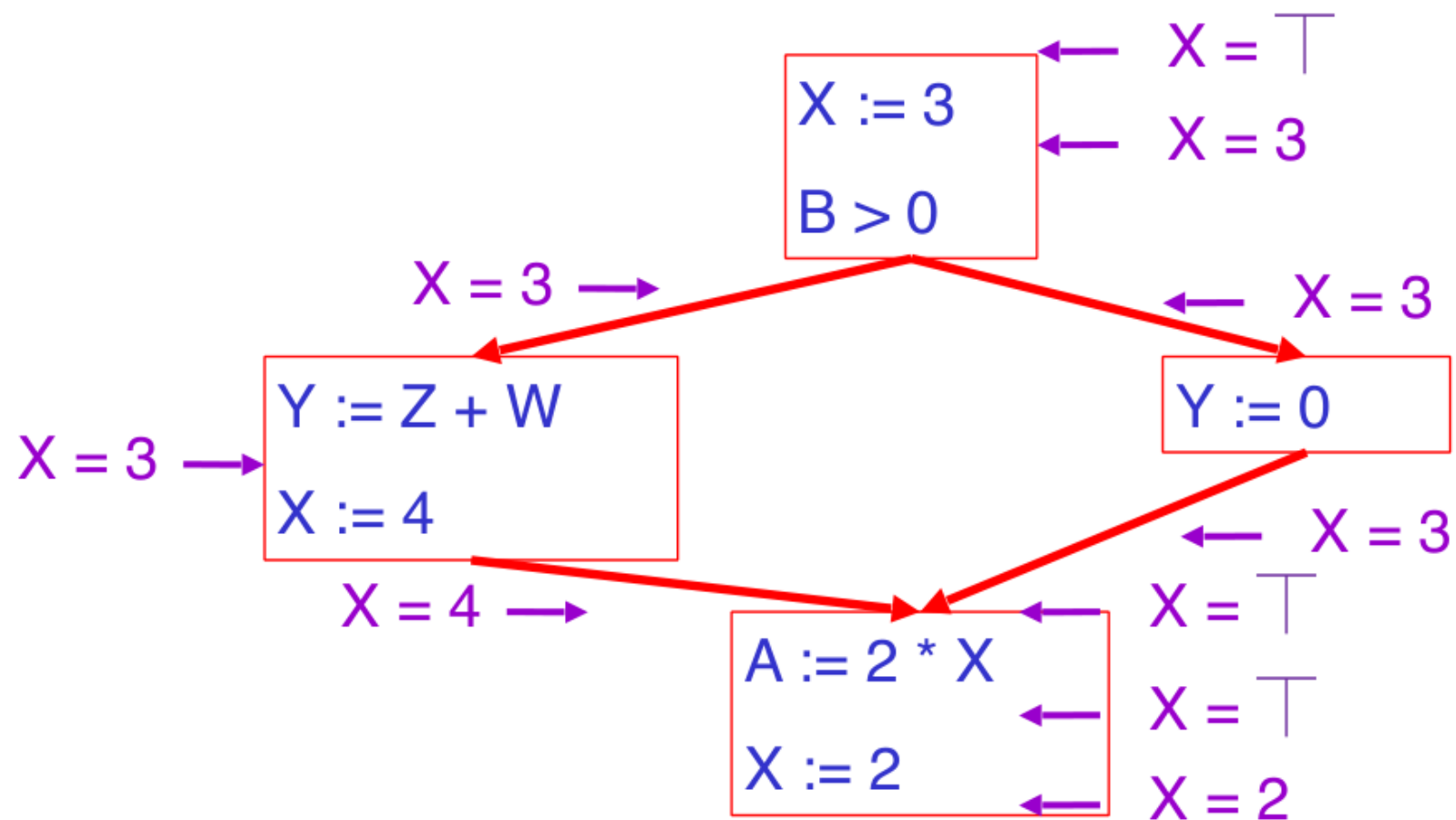


3.2 全局常量传播

- 采用下列标记

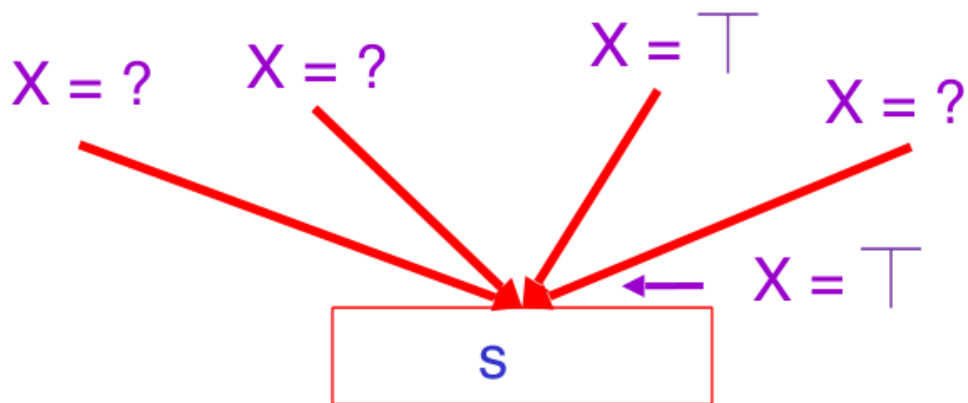
记号	含义
\perp	不会被执行
c	常数值c
T	非常数值

3.2 全局常量传播

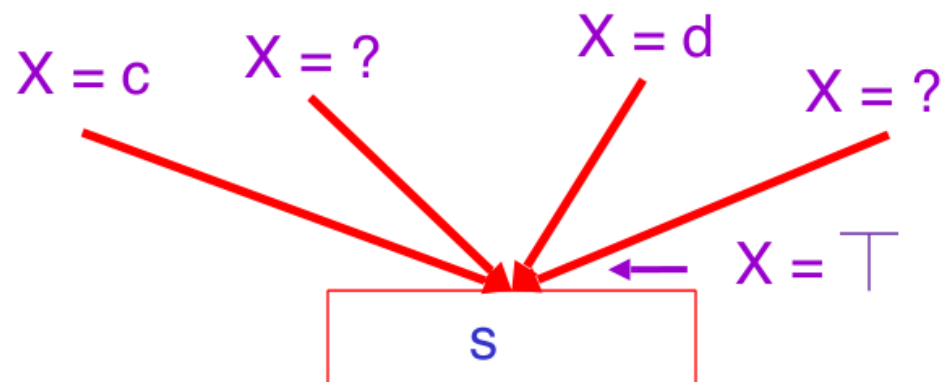


3.2 全局常量传播

- 计算规则



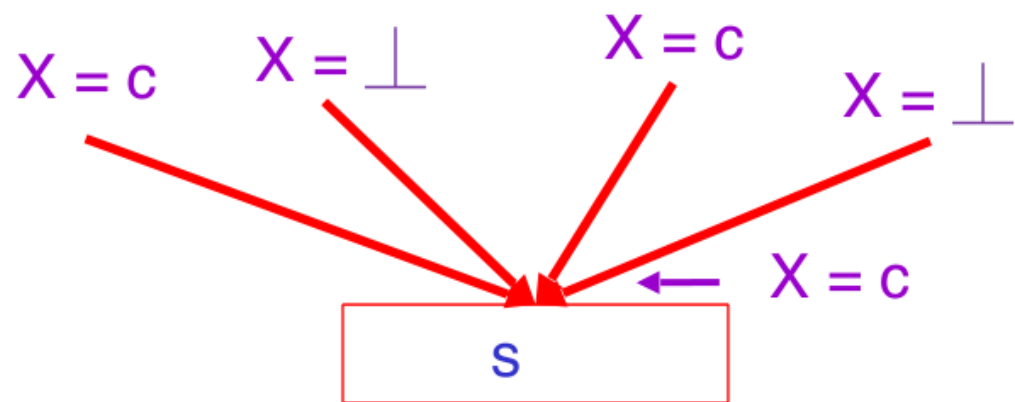
非变量 \Rightarrow 非变量



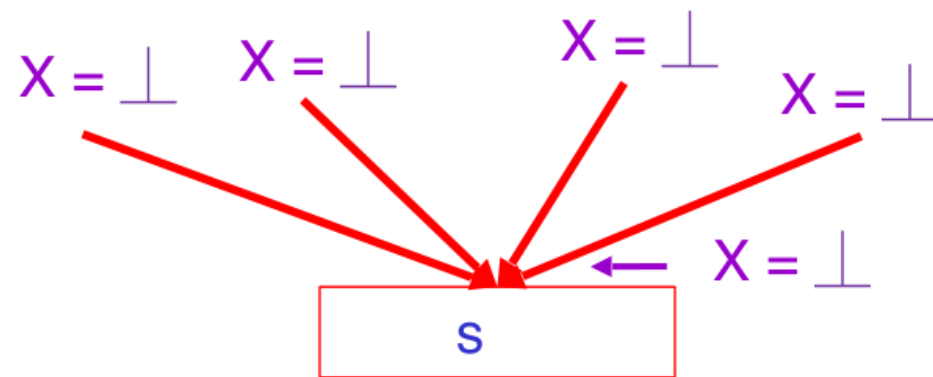
入边不同常量 \Rightarrow 非变量

3.2 全局常量传播

- 计算规则



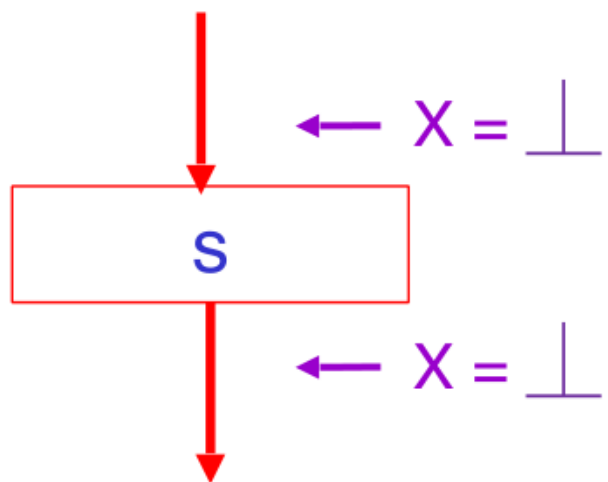
常量 \Rightarrow 常量



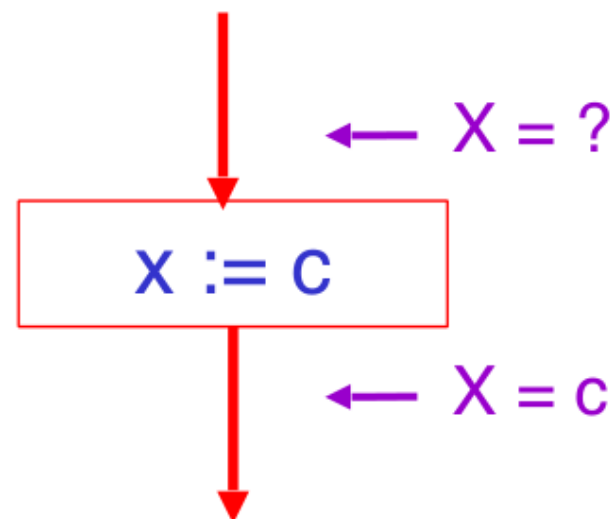
不可达 \Rightarrow 不可达

3.2 全局常量传播

- 计算规则

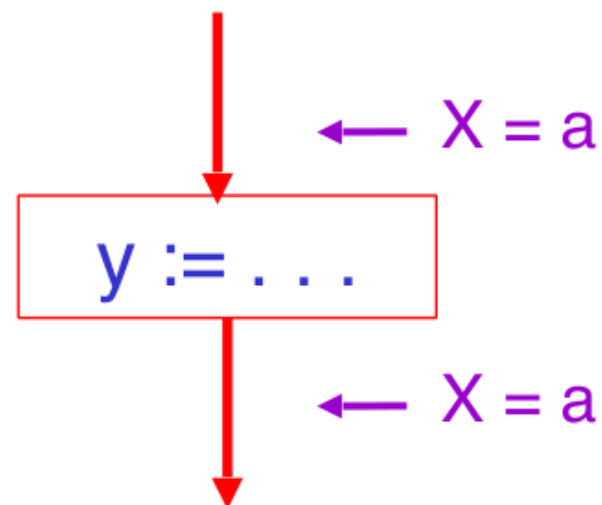
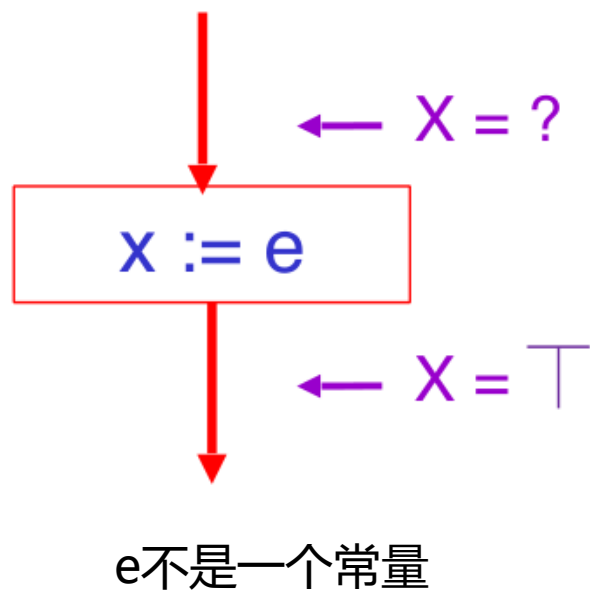


s不是对x赋值



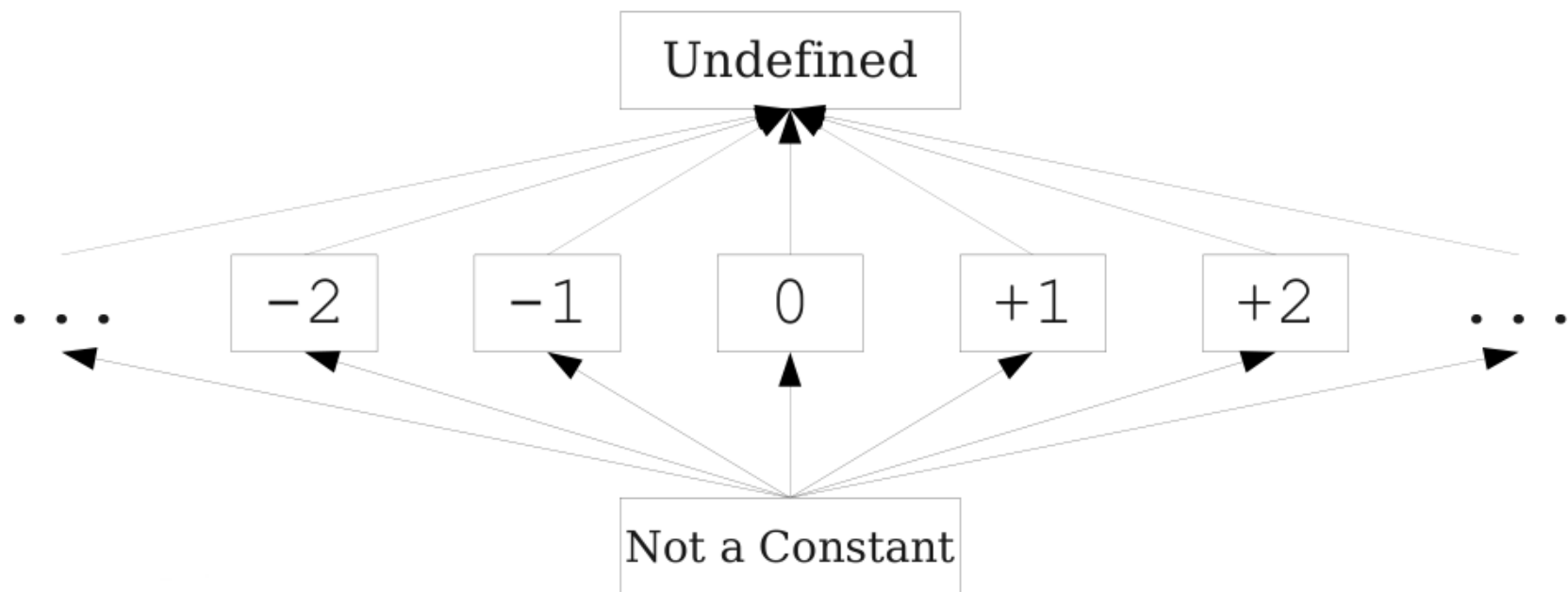
3.2 全局常量传播

- 计算规则



3.2 全局常量传播

- 数学基础
 - 交半格 meet semi-lattice



3.3 liveness analysis

- 明确变量使用范围
 - 比作用域更精细
- 推导方法
 - 类似于常量传播
 - 从后向前计算
 - live: true
 - dead: false

```
int f(int x) {  
    int a = x + 2;  
    int b = a * a;  
    int c = b + x;  
    return c;  
}
```

x is live

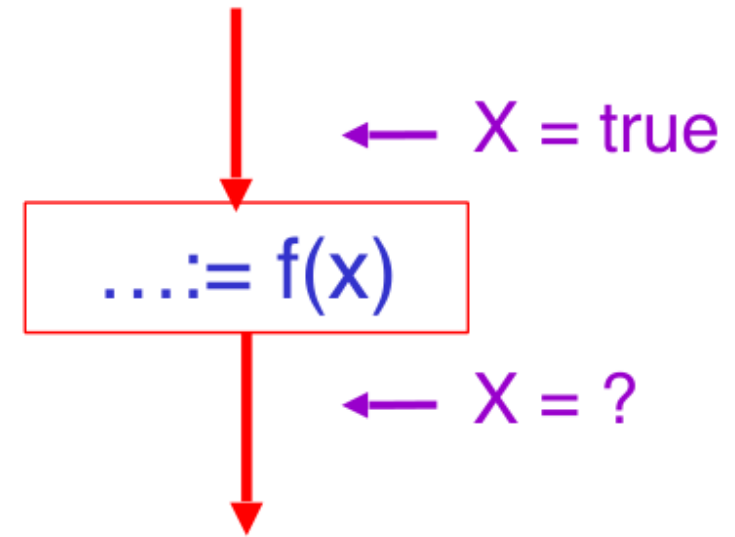
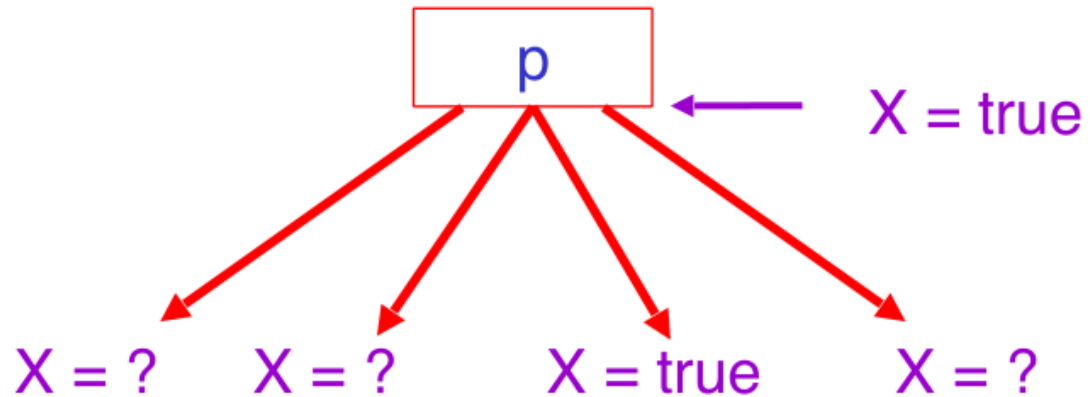
a and x are live

b and x are live

c is live

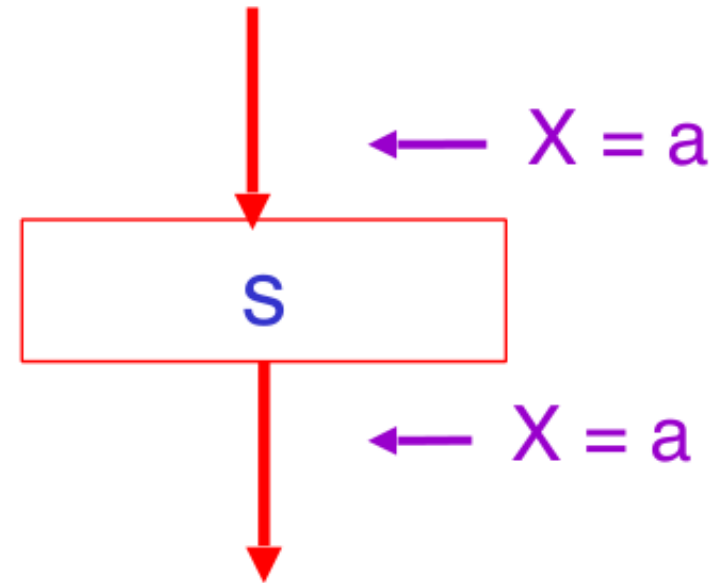
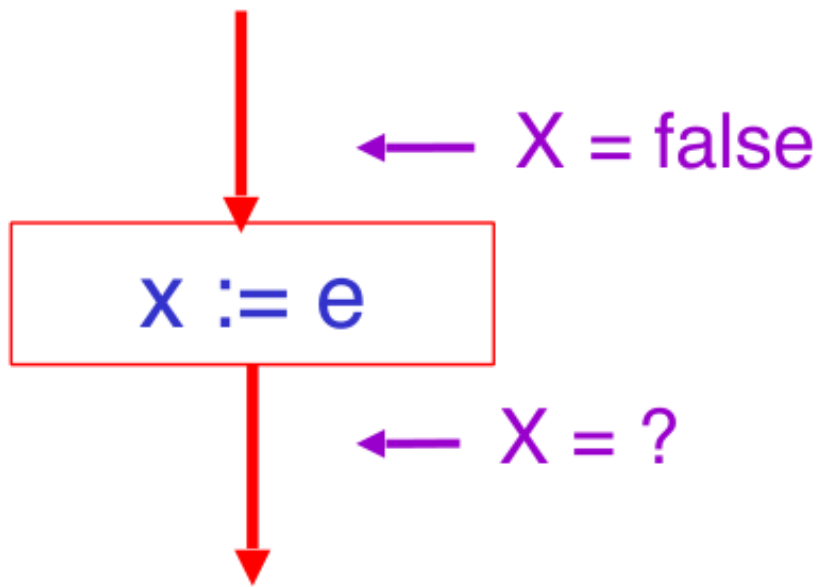
3.3 liveness analysis

- 计算规则



3.3 liveness analysis

- 计算规则



4.6 目标代码生成

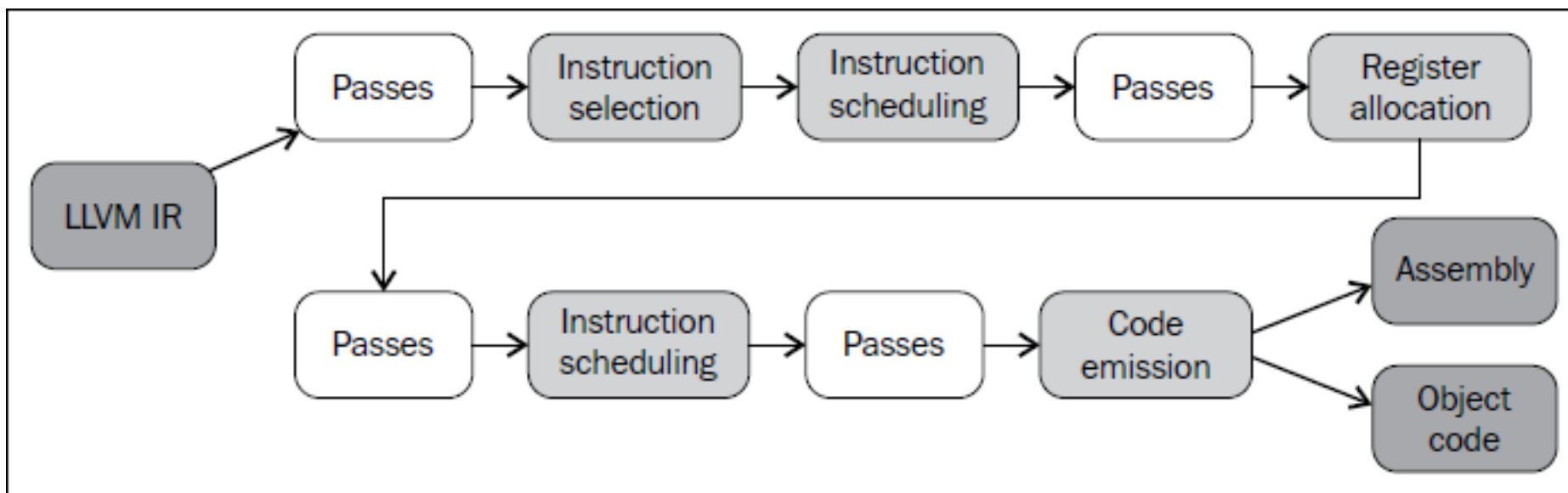
- 目标程序
 - 绝对机器代码
 - 可再定位机器代码
 - 汇编语言
- 指令集
- 寄存器数量

4.2 llvm-后端架构

- 英文文档
 - Getting Started with LLVM Core Libraries
 - Writing an LLVM Backend
 - The LLVM Target-Independent Code Generator
 - Tutorial: Creating an LLVM Backend for the Cpu0 Architecture
- 中文文档
 - 知乎

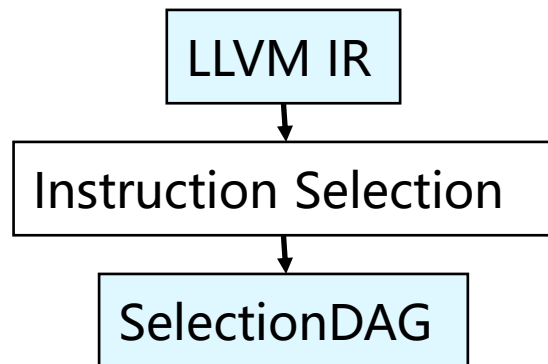
4.2 llvm-后端架构

LLVM IR -> 后端 -> 汇编/目标代码



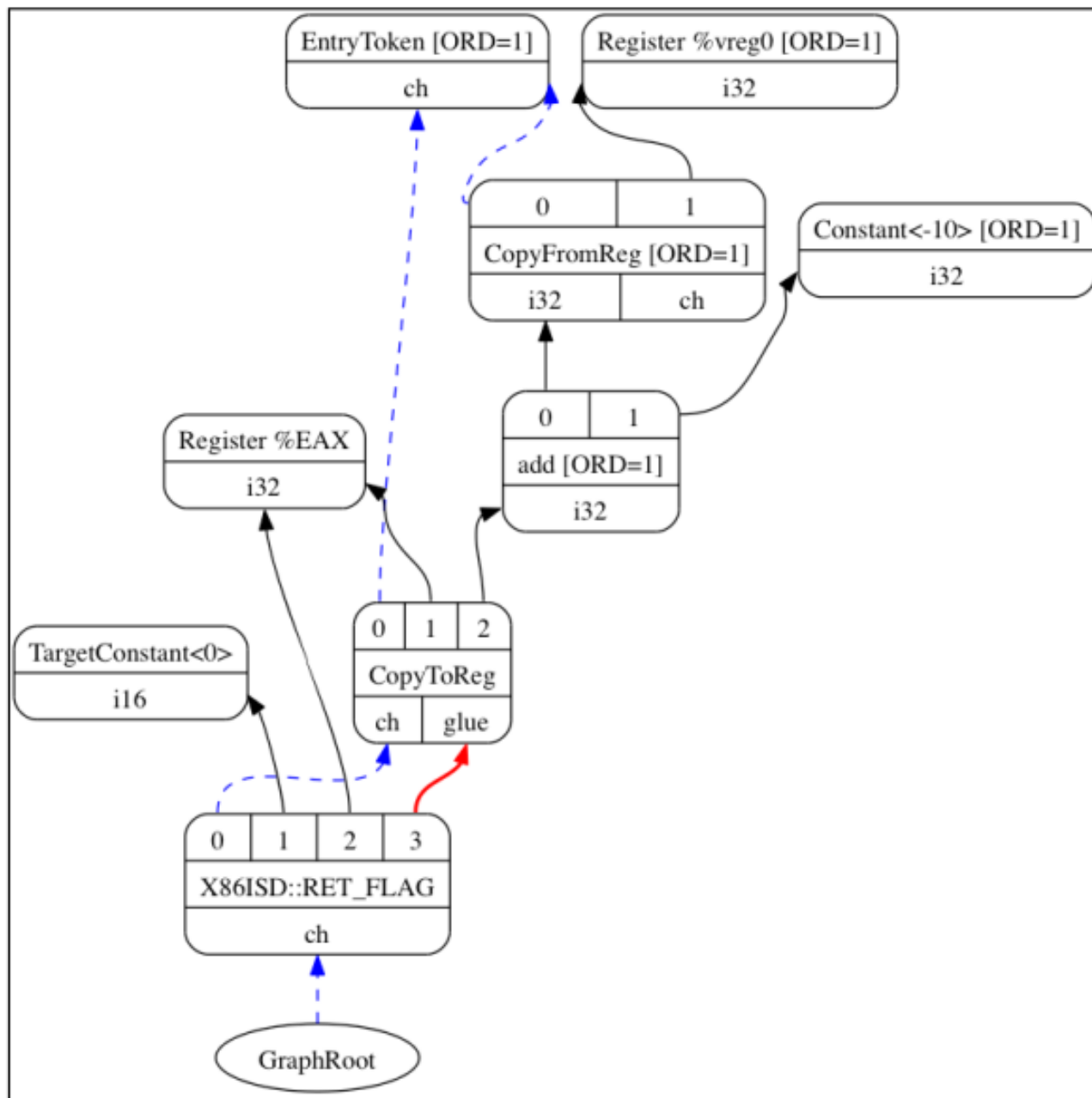
灰框是一些关键步骤，可能由多个pass组成，称为super pass
白框主要为一些优化的pass

4.2 指令选择



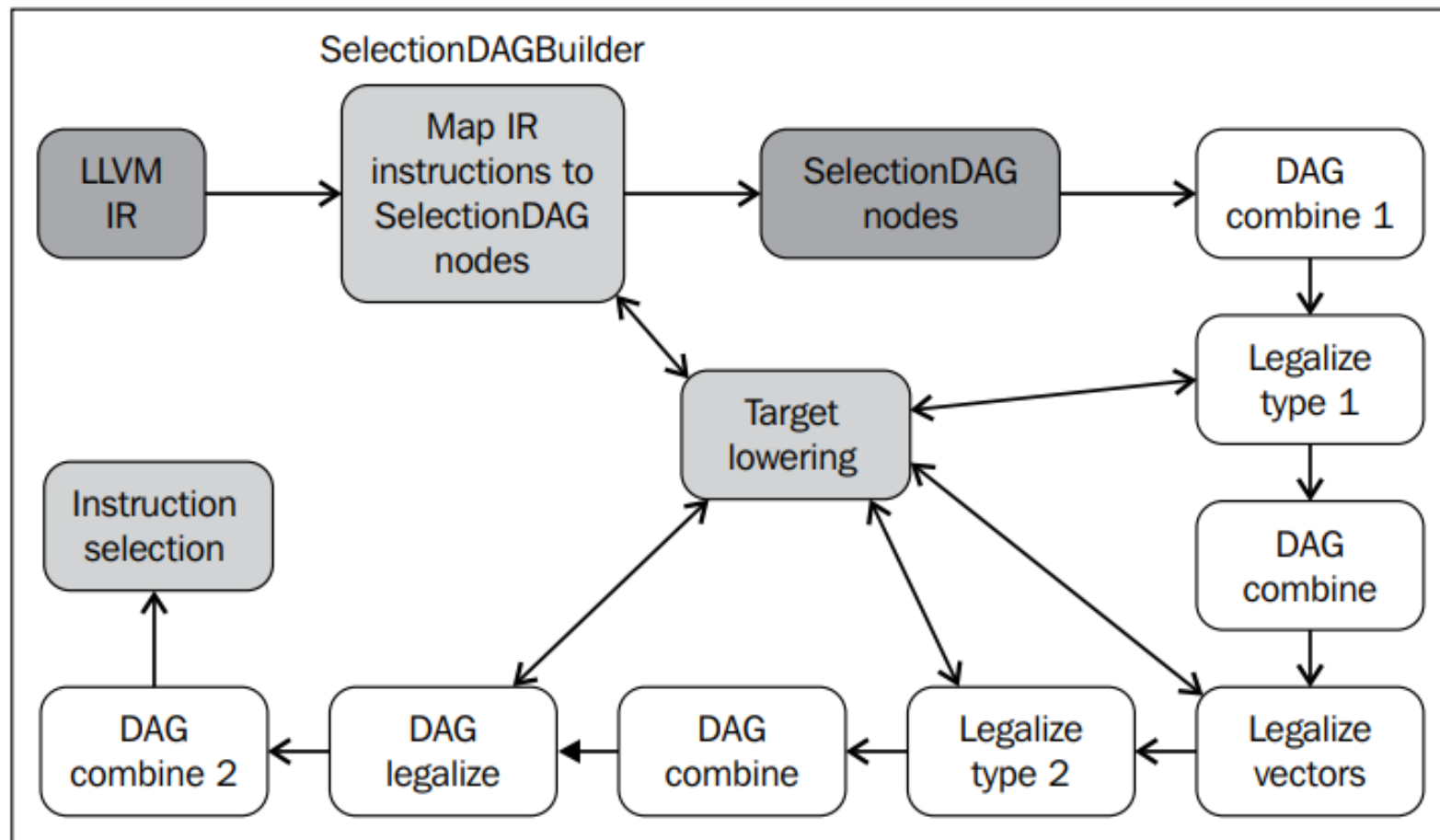
- SelectionDAG

- 基本上每个指令一个结点
- 黑线：数据依赖关系
- 蓝线：非数据依赖
- 红线：中间不能插入结点



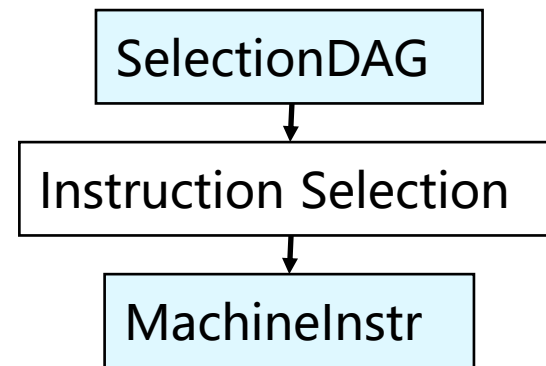
4.2 指令选择

- IR指令->ISA指令
- DAG合并
 - 优化
- 类型合法化
 - IR类型->ISA类型



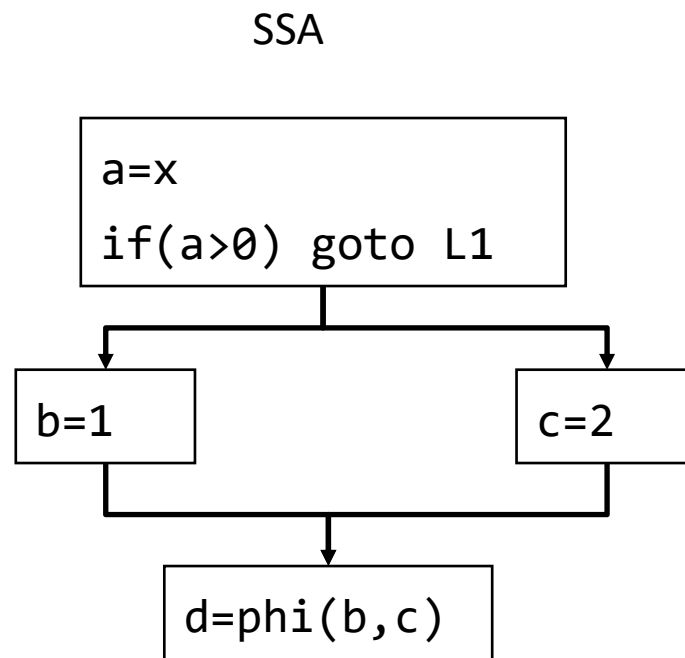
4.3 指令调度

- 分配前调度
 - 将DAG转换成指令序列
 - MachineInstr: 三地址代码
- 分配后调度
 - 主要是进行优化



4.4 寄存器分配

- 目标
 - 虚寄存器->物理寄存器
- 主要问题
 - phi函数: 转换为非SSA
 - 物理寄存器数量有限
 - 溢出(spill)



4.4 寄存器分配

- 目标
 - 虚寄存器->物理寄存器
- 主要问题
 - phi函数：转换为非SSA
 - 物理寄存器数量有限
 - 溢出(spill)

4.4 寄存器分配

- 分析liveness
- spilling
 - store-load
- coalescing
 - 减少复制
 - 增加冲突

```
int sum(char * v, int n) {
```

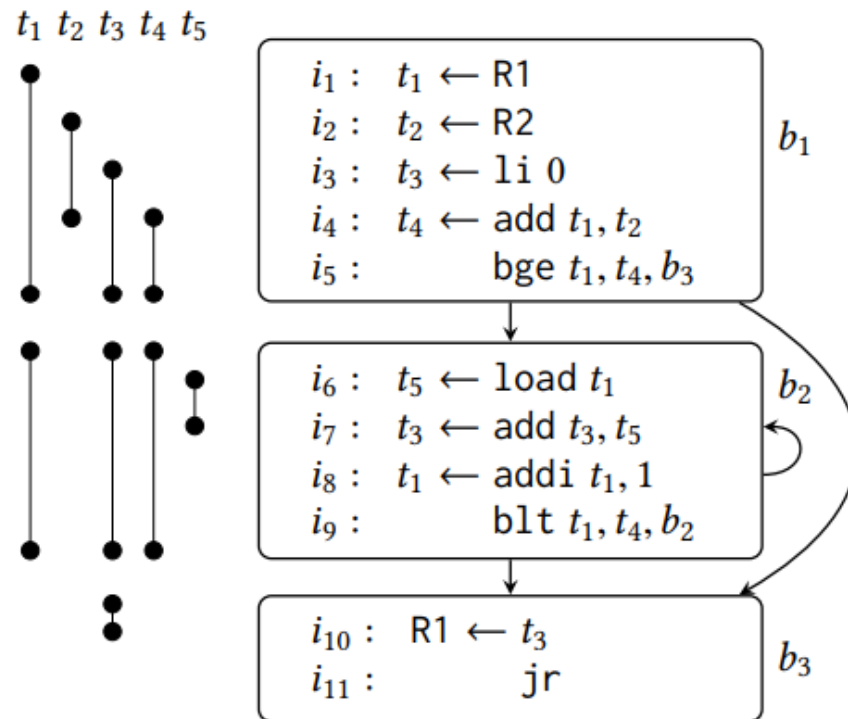
```
    int s = 0;
```

```
    for (int i = 0; i < n; i++) {  
        s += v[i];  
    }
```

```
    return s;
```

```
}
```

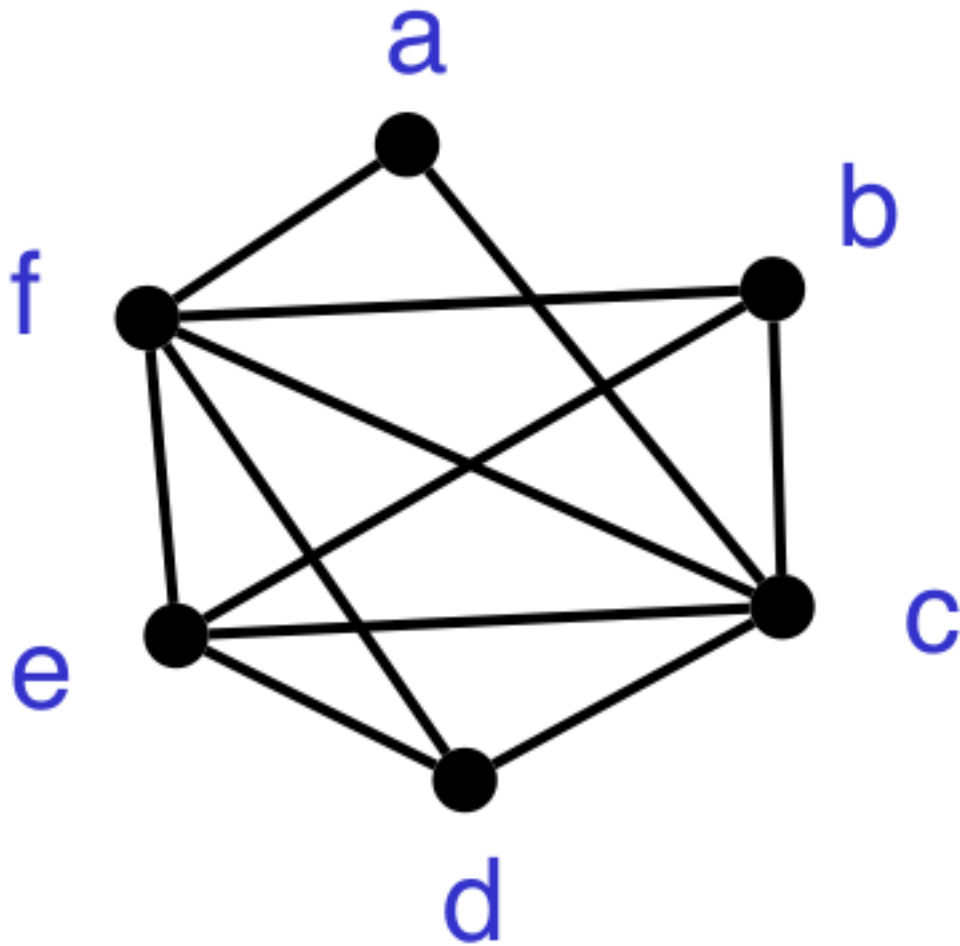
(a) C source code



(b) Live ranges and CFG

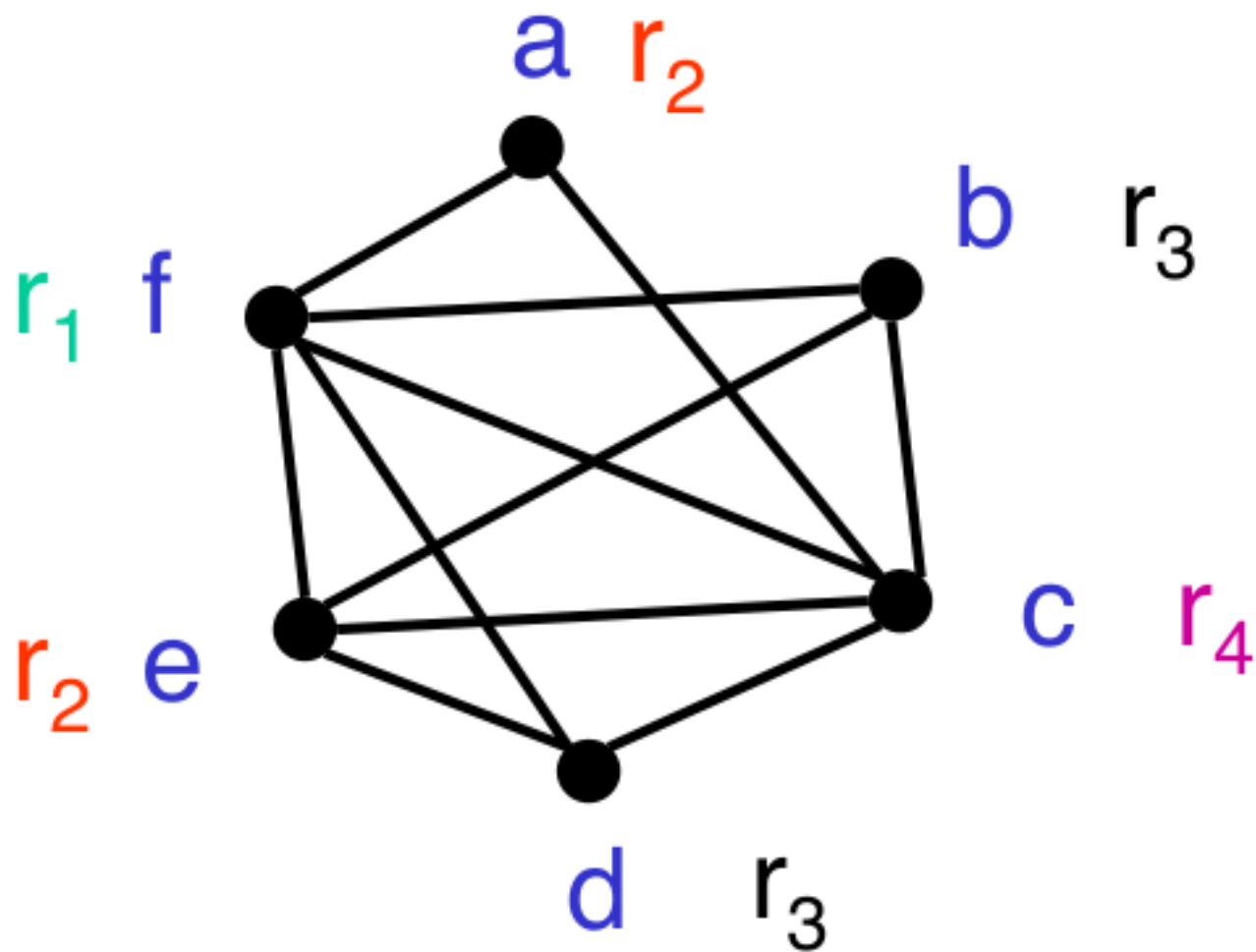
4.4 寄存器分配

- 图着色算法
 - register interference graph
 - 结点为变量
 - 边表示两个变量不能同寄存器
 - 邻接结点分配不同的颜色
 - NP-hard问题



4.4 寄存器分配

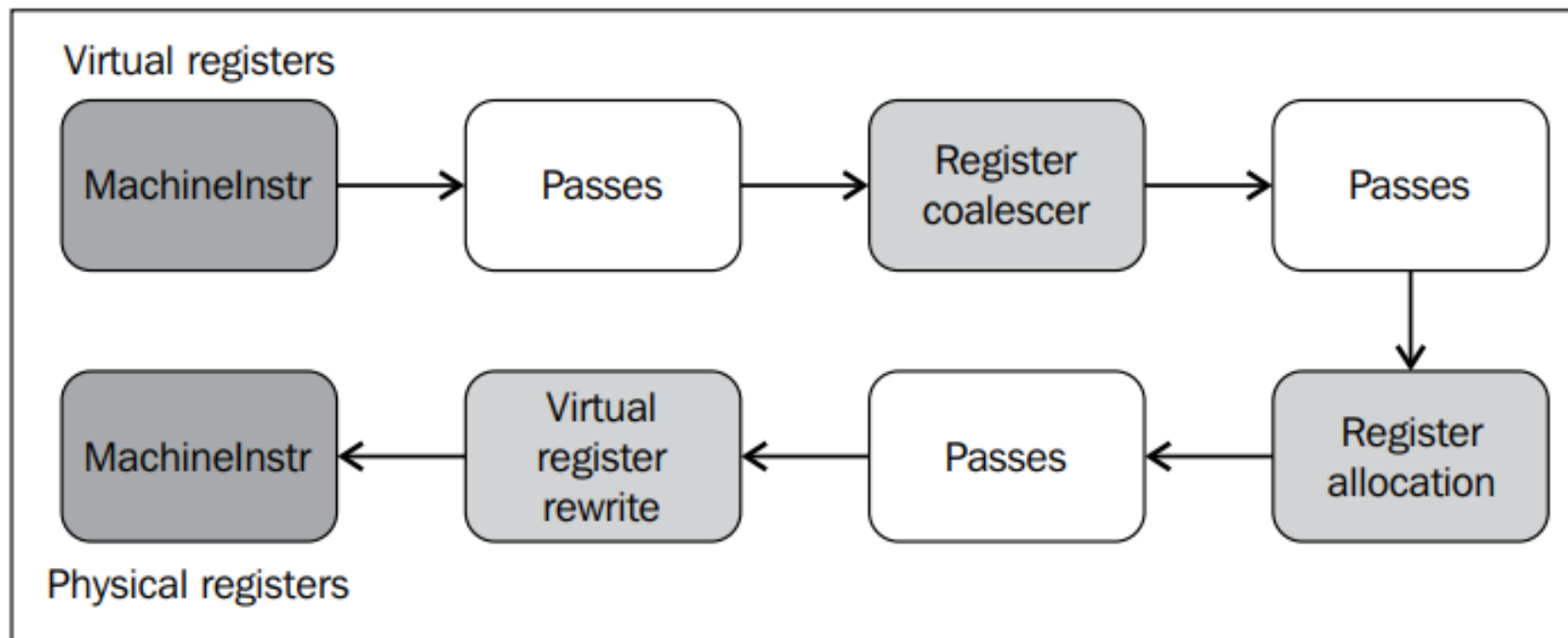
- 图着色算法
 - register interference graph
 - 结点为变量
 - 边表示两个变量不能同寄存器
 - 邻接结点分配不同的颜色
 - NP-hard问题



4.4 寄存器分配

- llvm所用算法

- fast
- basic
- greedy
- PBQP



4.5 代码输出

