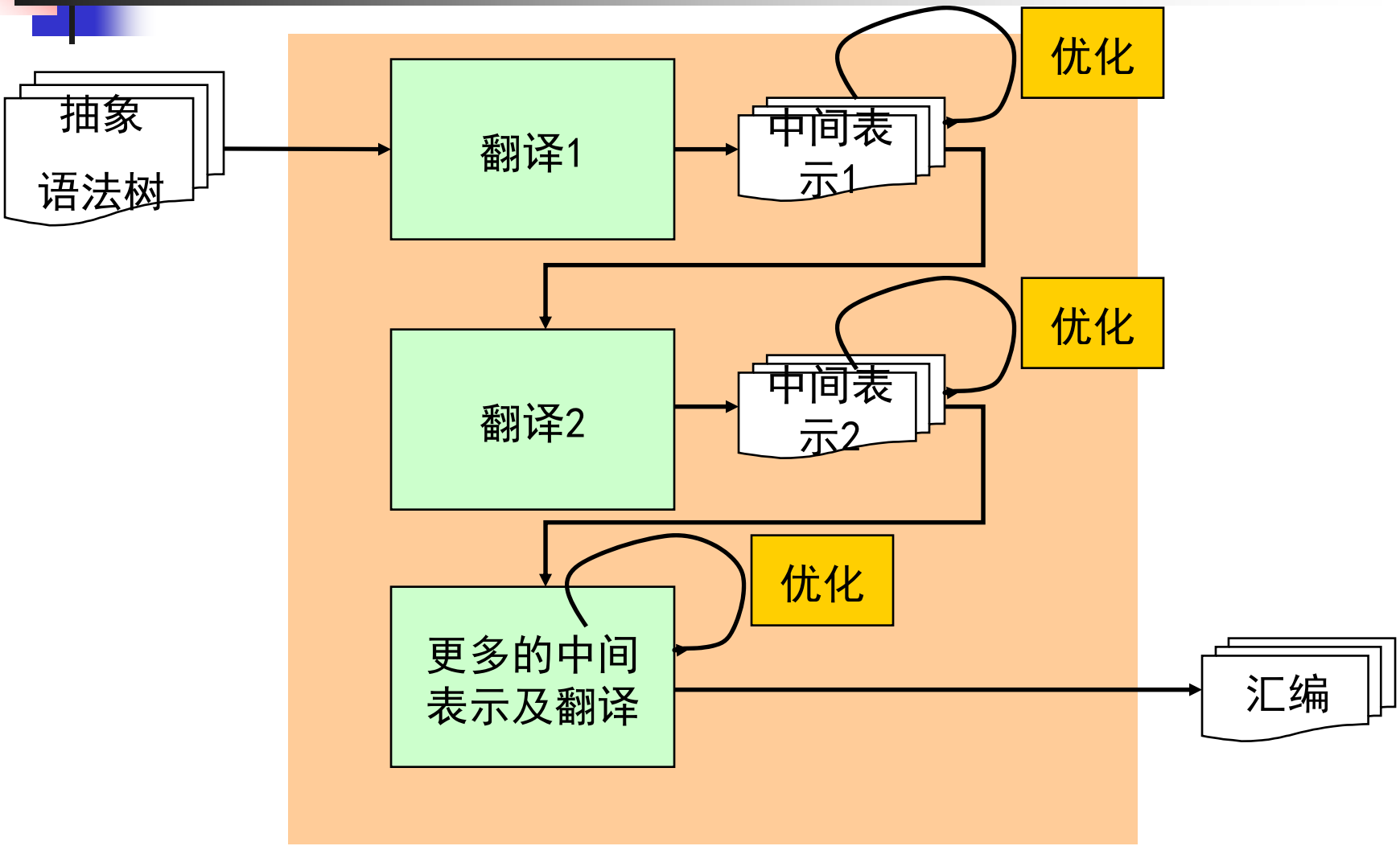
代码优化：中间表示上的优化

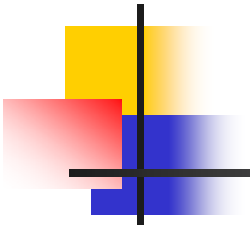
编译原理

华保健

bjhua@ustc.edu.cn

中间表示上优化的地位



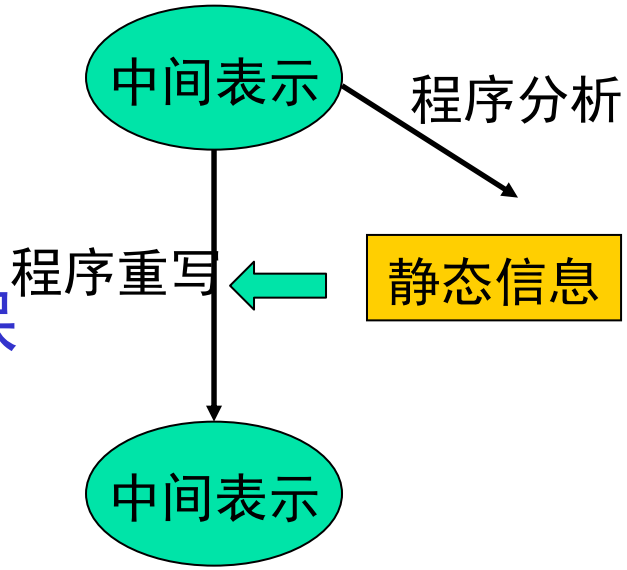


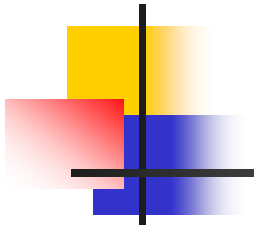
中间表示上的代码优化

- 依赖于具体所使用的中间表示：
  - 控制流图（CFG）、控制依赖图（CDG）、静态单赋值形式（SSA）、后续传递风格（CPS）等
- 共同的特点是需要进行程序分析
  - 优化是全局进行的，而不是局部
  - 通用的模式是：程序分析➔程序重写
- 在这部分中，我们将基于控制流图进行讨论
  - 但类似的技术可以用于其它类型的中间表示

优化的一般模式

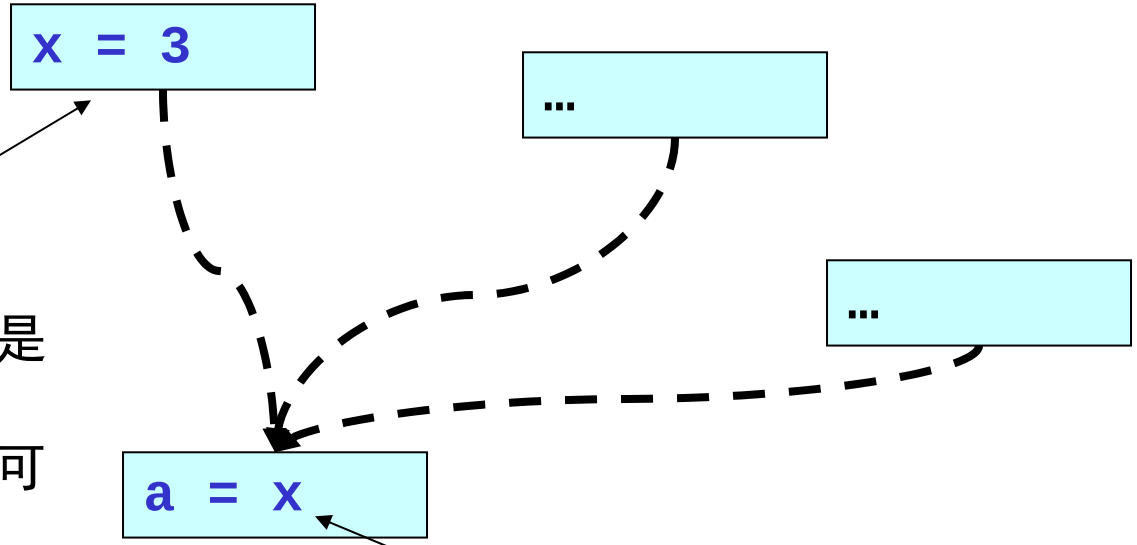
- 程序分析
  - 控制流分析，数据流分析， 依赖分析， …
  - 得到被优化程序的静态保守信息
    - 是对动态运行行为的近似
- 程序重写
  - 以上一步得到的信息制导对程序的重写





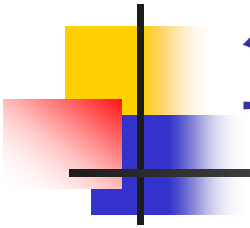
常量传播

常量传播



先进行到达定义分析，如果这个定义“x=3”是唯一能够到达使用“a=x”的定义，那么可以进行这个替换！

是否可以把x的使用替换成x的定义3？



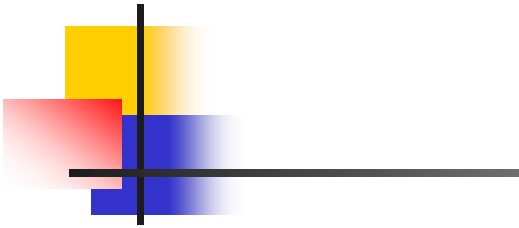
算法

```
// 常量传播算法
const_prop(Prog_t p)
  // 第一步：程序分析
  reaching_definition(p);
  // 第二步：程序改写
  foreach (stm s in p: y = x1, ..., xn)
    foreach (use of xi in s)
      if(the reaching def of xi is unique: xi = n)
        y = x1, ..., xi-1, n, xi+1, ..., xn
```

讨论

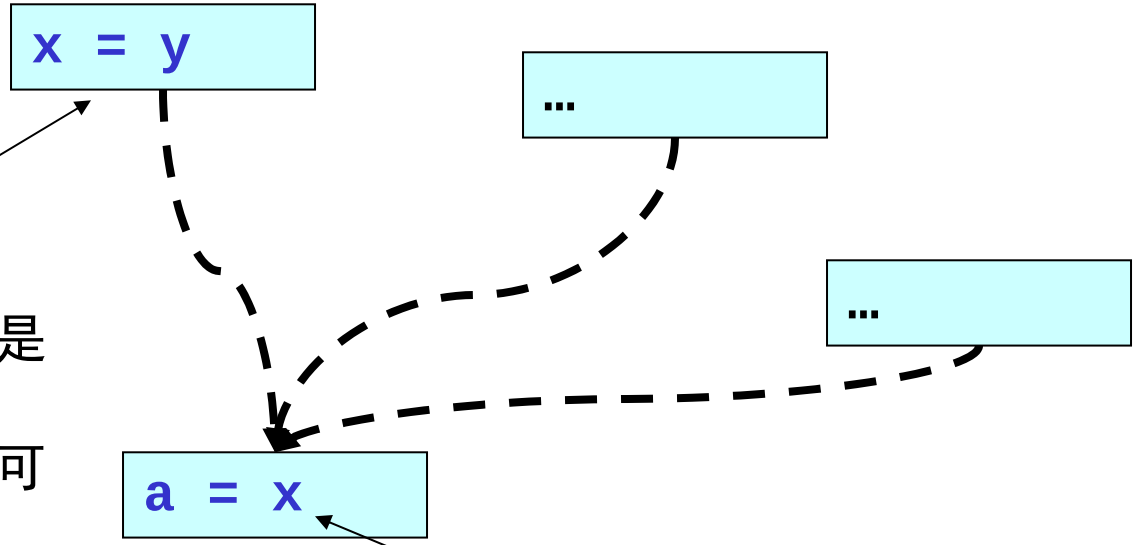
```
// 对示例程序
x = 1;
y = 2;
z = x + y;
a = z + 5;
print (a);
// 第一步：常量传播优化
x = 1;
y = 2;
z = 1 + 2;
a = z + 5;
print (a);
```

```
// 第二步：常量折叠优化
x = 1;
y = 2;
z = 3;
a = z + 5;
print (a);
// 第三步：常量传播优化
x = 1;
y = 2;
z = 3;
a = 3 + 5;
print (a);
```



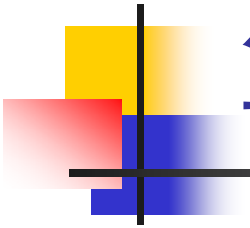
拷贝传播

拷贝传播



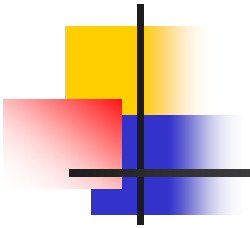
先进行到达定义分析，如果这个定义“x=y”是唯一能够到达使用“a=x”的定义，那么可以进行这个替换！

是否可以把x的使用替换成x的定义y？



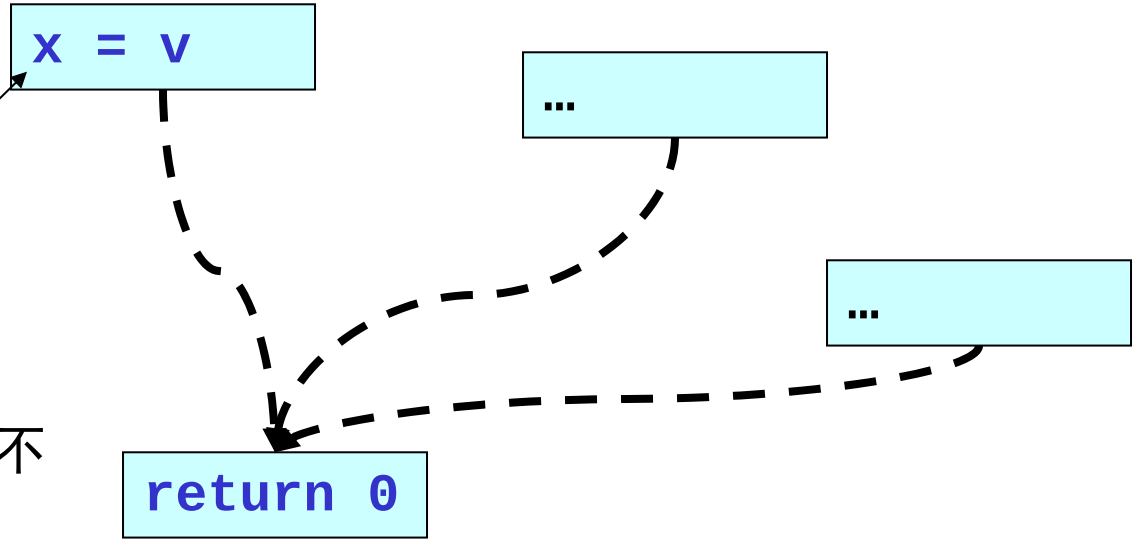
算法

```
// 拷贝传播算法
copy_prop(Prog_t p)
  // 第一步：程序分析
  reaching_definition(p);
  // 第二步：程序改写
  foreach (stm s in p: y = x1, ..., xn)
    foreach (use of xi in s)
      if(the reaching def of xi is unique: xi = z)
        y = x1, ..., xi-1, z, xi+1, ..., xn
```



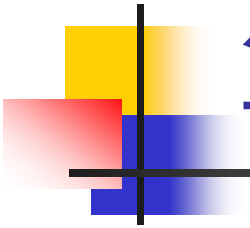
死代码删除

死代码删除



能把这个语句移除么？

进行活性分析，如果x不是该语句的live_out，则可以将该语句移除。



算法

```
// 死代码删除算法
dead_code(Prog_t p)
  // 第一步：程序分析
  liveness_analysis(p);
  // 第二步：程序改写
  foreach (stm s in p: y = ...)
    if (y is NOT in live_out[s])
      remove (s);
```

讨论

```
// 对示例程序
x = 1;
y = 2;
z = x + y;
a = z + 5;
print (a);
// 第一步：常量传播优化
x = 1;
y = 2;
z = 1 + 2;
a = z + 5;
print (a);
```

```
// 第二步：常量折叠优化
x = 1;
y = 2;
z = 3;
a = z + 5;
print (a);
// 第三步：常量传播优化
x = 1;
y = 2;
z = 3;
a = 3 + 5;
print (a);
```