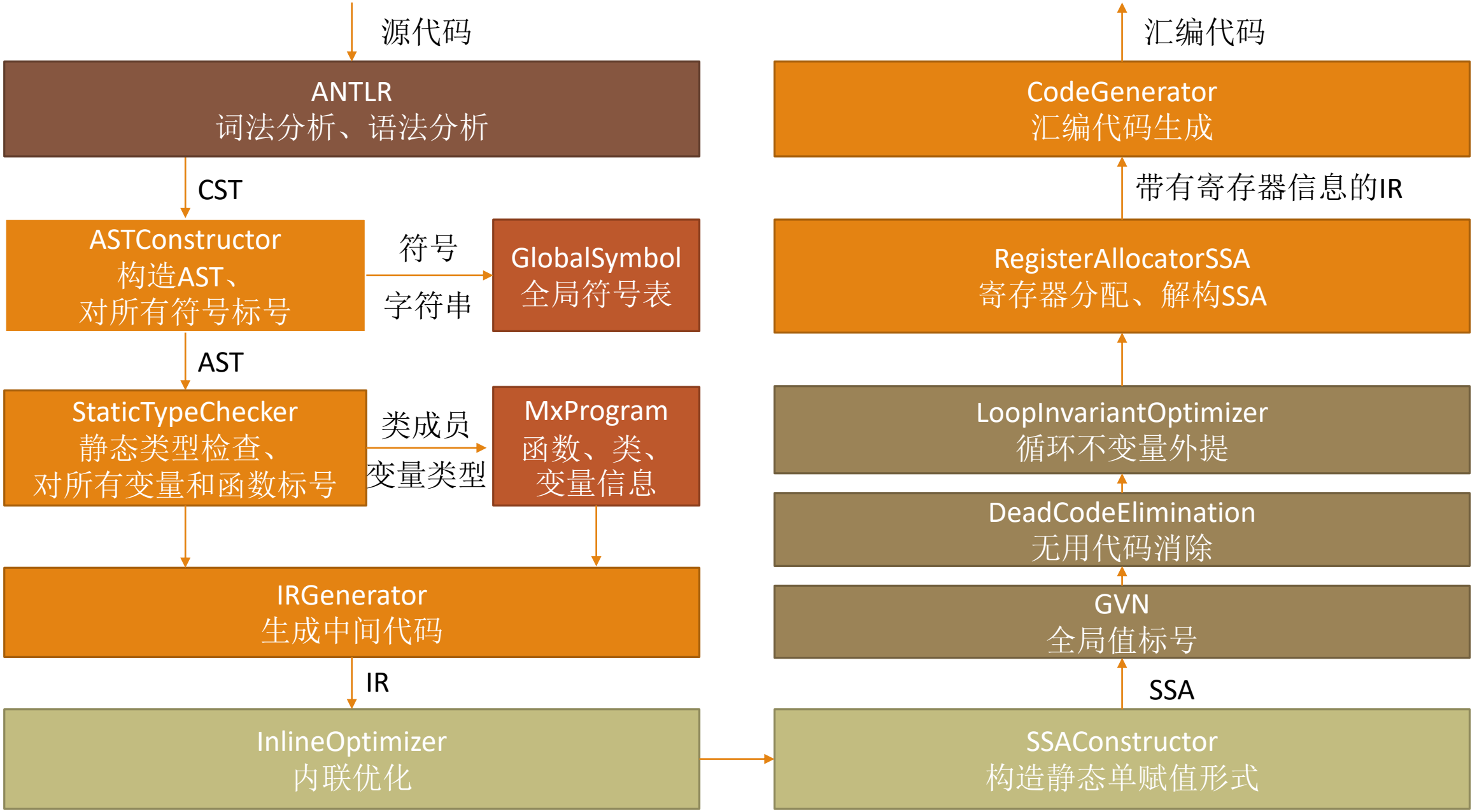
COMPILER 2017

张哲恺


源代码
ANTLR
词法分析、语法分析
CST
ASTConstructor
构造AST、
对所有符号标号
符号
字符串
GlobalSymbol
全局符号表
AST
StaticTypeChecker
静态类型检查、
对所有变量和函数标号
类成员
变量类型
MxProgram
函数、类、
变量信息
IRGenerator
生成中间代码
IR
InlineOptimizer
内联优化
SSAConstructor
构造静态单赋值形式
SSA
GVN
全局值标号
DeadCodeElimination
无用代码消除
LoopInvariantOptimizer
循环不变量外提
RegisterAllocatorSSA
寄存器分配、解构SSA
带有寄存器信息的IR
CodeGenerator
汇编代码生成
汇编代码

前端

使用ANTLR 4进行词法分析和语法分析，得到CST

遍历CST，将所有符号标号，并建立AST

- 设计模式：AST支持Visitor模式和Listener模式进行访问

静态代码检查，对所有的函数、类、全局变量、局部变量进行标号

- 标号后名称相同的不同变量得到区分
- AST上的每一个变量结点（ASTExprVar）都指明了它是局部变量、成员变量或全局变量

中间代码生成

中间代码：

- 每个函数都是一个控制流图
- 每一个代码块都是控制流图上的一个节点，一个代码块有若干个前驱块和至多两个后继块
- 代码块内采用三地址码对指令进行编码
- 控制流图有且仅有一个入口块和一个出口块

遍历AST进行翻译过程

- 每个语句都被翻译成有一个入口块和一个出口块的子图

基于引用计数的简易垃圾回收：

- 将所有的值区分为左值(lvalue)、将亡值(xvalue)和纯右值(rvalue)
- 当左值或将亡值赋给一个变量时引用计数加一
- 每条语句结束后所有的将亡值的引用计数减一

代码优化

我自己的数据：SHA-1计算和暴力破解

频繁使用小函数

使用较多的局部变量

存在多次执行的重复计算

存在频繁执行的循环次数固定的小循环

```
for(i=0;i<length;i++)
    chunks[i/64][i%64/4] = chunks[i/64][i%64/4] | (input[i] << ((3-i%4)*8));
chunks[i/64][i%64/4] = chunks[i/64][i%64/4] | (128 << ((3-i%4)*8));
```

```
for(j=16;j<80;j++)
    chunks[i][j] = rotate_left(chunks[i][j-3] ^
```

```
for(j=16;j<80;j++)
    chunks[i][j] = rotate_void p

int a = h0;                    & 655
int b = h1;                   5535)
int c = h2;                    & 655
int d = h3;
int e = h4;
for(j=0;j<80;j++)
{
    int f;
    int k;
```

代码优化

我自己的数据：SHA-1计算和暴力破解

频繁使用小函数：内联

使用较多的局部变量：寄存器分配

存在多次执行的重复计算：冗余消除

存在频繁执行的循环次数固定的小循环：循环展开

其他数据上的典型现象：

- 无用代码
- 循环不变量

```
void cost_a_lot_of_time()
{
    int a = 3100;
    int b = 0;
    int c = 1;
    for (b = 0; b < 100000000; ++b)
        c = c * 2 - c;
    println(toString(a));
}

for(g = 0; g < n; ++g)
{
    bool t1 = (((a == b) && c > 0)
    bool t2 = (((a == b) && c > 0)
    bool t3 = (((a == b) && c > 0)
    bool t4 = (((a == b) && c > 0)
    bool t5 = (((a == b) && c > 0)
    bool t6 = (((a == b) && c > 0)
```

代码优化

实现的优化：

- 函数内联
- 基于全局值标号的冗余发现
- 无用代码消除
- 循环不变量外提
- 基于SSA的寄存器分配

内联优化

函数调用有一定的开销，在小函数上调用开销占比较大

越小的函数越有可能通过内联而获利

算法：

- 每次选取最小的函数进行内联
- 非递归函数内联完毕后即可删除
- 允许递归函数进行内联，内联一次后重新加入待内联函数的集合
- 采用优先队列维护最小的函数
- 直到最小的函数大小超过一定值，算法结束

评估函数大小：

- 我的方法：
  - 函数大小=指令数+块个数^2　　（此函数不调用其他函数）
  - 函数大小=指令数+块个数^2+常数　　（此函数仍然调用其他函数）

冗余代码发现

一个值如果在之前被计算过，我们希望它不被重复计算：

```
x = a + b
y = a
z = y + b
```

单个基本块内：局部值标号

对每个值计算一个哈希值：

- 相同的常数拥有相同的哈希值
- 两个表达式如果操作数相同，操作符相同，也拥有相同的哈希值
- 每个不可预测（如外部函数调用）的值赋予一个单独的哈希值
- 例如，对于二元运算符op, 有Val(x op y) = Hash(op, Val(x), Val(y))

拥有相同哈希值的值认为它们相等

哈希冲突

- 使用较强的哈希函数，例如密码学中的SHA系列函数
- 将每个值作为结点、操作数作为子节点构成树结构，比较两个结点时递归进行比较

冗余代码发现

将范围扩展到全局（整个函数），存在的困难：

- 变量的值可能发生变化，即使两个表达式完全相同我们也不能断定它们的值相同
- 在某些执行路径上我们所需要的之前计算的值可能被覆盖，也可能没有被执行

静态单赋值形式(SSA)：

- 每一个变量只在定义时被赋值一次
- 每个变量的活动范围是控制流图上被它支配的范围
- 在多个变量汇合处使用ϕ函数进行合并

在SSA上可以顺利进行全局范围内的值标号

全局值标号

对ϕ函数进行标号：

- x_4 = ϕ(pred1: x_1, pred2: x_2, pred3: x_3)
- y_4 = ϕ(pred1: y_1, pred2: y_2, pred3: y_3)
- 可以看到同一个基本块内的ϕ函数是相互关联的
- 当程序的执行路径是由pred1块到达当前块时，x_4为x_1，y_4取值为y_1
- 当程序的执行路径是由pred2块到达当前块时，x_4为x_2，y_4取值为y_2
- ……
- 我们可以给每个基本块的所有ϕ函数一个相同的ID，即

Val(ϕ(x1, x2, x3)) = Hash(block_id, Val(x1), Val(x2), Val(x3))

全局值标号

扩展：

一些运算符具有交换律： a op b == b op a

- 计算哈希值可以按照操作数的哈希值进行排序，即

$$\mathrm{Val}(a\ \mathrm{op}\ b) = \begin{cases} \mathrm{Hash}\big(\mathrm{op}, \mathrm{Val}(a), \mathrm{Val}(b)\big) & (\mathrm{Val}(a) > \mathrm{Val}(b)) \\ \mathrm{Hash}\big(\mathrm{op}, \mathrm{Val}(a), \mathrm{Val}(b)\big) & (\mathrm{Val}(\mathrm{a}) \le \mathrm{Val}(\mathrm{b})) \end{cases}$$

一些运算符具有结合律： (a op b) op c == a op (b op c)

- 可以把连续的具有结合律的相同运算合并，即

$$\mathrm{Val}(a\ \mathrm{op}\ b\ \mathrm{op}\ c) = \mathrm{Hash}(\mathrm{op}, \mathrm{Val}(a), \mathrm{Val}(b), \mathrm{Val}(c))$$

运算的其他性质，例如 -(-a) == a

- 利用之前提到的树结构，我们保留了操作数的具体内容，我们可以做到

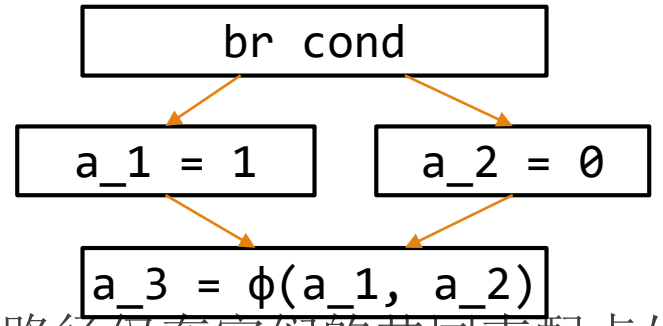
$$\mathrm{Val}(\mathrm{neg}\ \mathrm{neg}\ a) = \mathrm{Val}(a)$$

全局值标号

发现相同的控制结构：

```
if(cond) a = 1; else a = 0;
if(cond) b = 1; else b = 0;
```

我们希望发现a和b在if语句之后具有相同的值



沿着ϕ函数的两个操作数对应的前驱块向上走，如果两条路径仅在它们的共同支配点处汇合，那么这个ϕ函数相当于条件选择语句 cond ? a_1 : a_2，其哈希值为Val(cond ? a : b) = Hash(op, Val(cond), Val(a), Val(b))

* 注：实现中利用了这个条件的充分不必要条件，考虑ϕ(pred1: x_1, pred2: x_2)，pred1和pred2的最近公共支配点为lca，支配树上pred1对应的lca的子节点为son1，pred2对应son2，son1 != son2，如果pred1反向支配son1，且pred2反向支配son2，可以推得两条路径仅在lca处汇合

全局值标号

标号顺序

我们希望一个表达式进行标号时其操作数均已被标号

使用逆后序(reverse postorder)遍历，在无循环时可以保证每个块在访问时其前驱块都被访问过，在有循环时，如果一个块有前驱未被访问， 它一定是一个循环头部

若一个表达式的操作符未被标号，我们给这个表达式一个唯一的哈希值

无用代码消除

如果一个变量从未被使用，且定义它的指令无副作用，我们可以将其删除

```
x = a * b;
```

```
x = a + b;
```

在SSA上我们可以很容易地发现一个变量的使用情况

删除无用的控制块：

- 删去无用的指令，我们可能还留下了一些无用的控制块，例如

  ```
  for(int i=0; i<1000; i++);
  ```

- 通常无需担心删去一个死循环（也并没有通用的方法判断循环停止：停机问题），在C/C++中，无副作用的循环是被允许删除的*

IR上并没有显式的循环、判断等结构，要想确定哪些块被删除可能比较困难

* https://stackoverflow.com/questions/3592557/optimizing-away-a-while1-in-c0x/

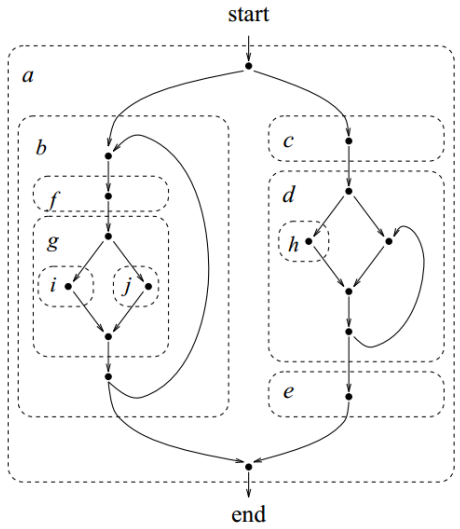
重新发现程序结构——PST

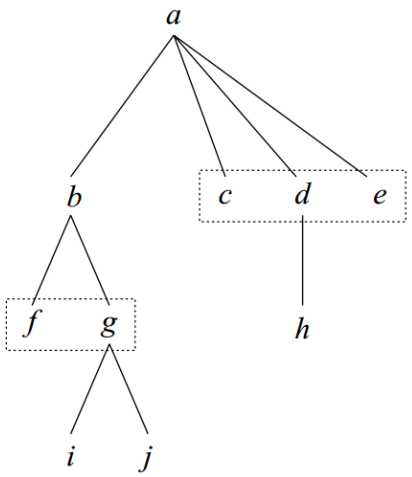
我们希望在IR上重新构建出程序的层次结构

可以使用程序结构树(Program Structure Tree, PST)

PST上的每一个节点对应控制流图上的一个单入单出块(Single Entry Single Exit, SESE)

如果一个SESE块中定义的变量都没有被外部使用，并且其中没有包含带有副作用的指令，我们就可以安全地删除这个块





循环不变量外提

```
for(int i=0; i<strlen(str); i++) { /*do something*/ }
```

strlen是一个代价不小的函数(O(n))，如果我们知道str不会随循环改变，strlen是无副作用的函数，并且对于相同的参数都返回相同的值，它可以被提到循环外面

```
size_t size = strlen(str);
for(int i=0; i<size; i++) { /*do something*/ }
```

为此我们定义一个值z = f(x)为循环不变量，当且仅当：

- x是循环不变量
- f(x)无副作用且返回值只依赖于x的值(constexpr)

在SSA上，由于ϕ函数的值依赖于执行路径，我们不认为它是循环不变量

循环不变量外提

利用循环检测(Loop Detection)发现IR中的循环，标记其循环头(loop header, loop preheader)

我们乐观地假定所有变量都是循环不变量，每个变量作为一个结点向依赖它的变量连一条边，得到一个依赖图

我们标记所有已知的非循环不变量的结点，并以此为起点标记所有能够到达的结点

删除这些标记的结点，剩下的图是一个DAG，进行拓扑排序依次将它们提到循环外

循环不变量外提

扩展：

我们希望循环中的一些不变的控制块也能被提到循环外部，例如

```
for(int i=0; i<N; i++) {
  int s = 0;
  for(int j=0; j<100; j++) s = s + j;
  sum = sum + s;
}
```

以及

```
for(int i=0; i<N; i++) {
  bool b = a && b && c;
  if(b) ans++;
}
```

循环不变量外提

方便起见，我们处理的仍然是SESE块

定义循环不变块为满足以下条件的SESE块：

- 所有的指令都是constexpr
- 使用的所有外部操作数都是循环不变量

由于这些块外提以后可能会有新的循环不变量被发现，我们定义循环不变量为

- (由一个循环不变块所定义的变量) 或者 (满足之前的定义条件)

我们把这些SESE块也纳入之前的图中进行处理即可

循环不变量外提

一点小问题：

```
for(int i=0; i<N; i++)
  if(d > 0)
    ans += c / d;
```

如果我们把c/d提到循环外，它就不会受到d > 0的保护，可能会使程序出现异常

解决方案：

1. 不把可能引发异常的指令提到循环外
2. 在每个被提出的可能引发异常的指令前加上保护条件：

```
int result = d != 0 ? c / d : 0;
for(int i=0; i<N; i++)
  if(d > 0)
    ans += result;
```

SSA上的寄存器分配

假设一段程序只有一个基本块，并且每个变量只被定义一次

如果在任意指令处寄存器压力（Register Pressure, 同时活动的变量数）不超过物理寄存器的个数，那么我们从前往后给每个变量分配一个寄存器，我们总是能够找到空闲的寄存器

SSA的性质：每个变量只在它的支配区域活动

如果我们把每条指令看作一个基本块构造支配树的话，它的活动区间就是以它为根的子树

而一个子树的寄存器分配情况并不影响到其他的子树

直觉上讲，在SSA上的寄存器分配就是线性分配的直接扩展

SSA上的寄存器分配

事实上，我们可以证明，SSA上变量的干涉图(Inference Graph)是弦图，其完美消除序列(Perfect Elimination Order, PEO)是支配树的后序遍历

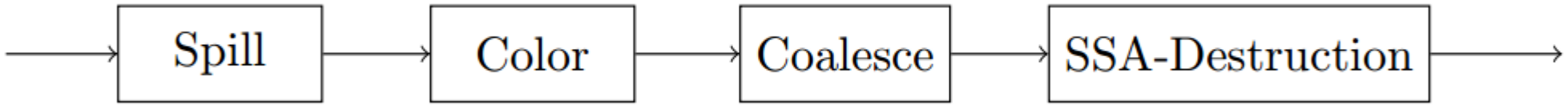
对于一个弦图，我们按照其完美消除序列的逆序进行贪心染色，所需要的色数就是弦图的最小色数

*弦(Chord)：连接环中不相邻的两个点的边

*弦图(Chord Graph)：任意长度大于3的环都至少有一条弦的无向图

*完美消除序列：对于序列中的每一个点，排在它后面的与之相连的点都是一个团

SSA上的寄存器分配



SSA上的寄存器分配

处理寄存器逐出(Spill)

对每个基本块进行逐出，维护当前在寄存器中的变量(W集合)，当寄存器不足时，选取距离下次使用最远的变量进行逐出(Min Algorithm)

- 由于循环通常会执行多次，如果一个变量的下一次使用是在循环外面，其下次使用距离需要加上一个常数。实现上，我们对每个离开循环的控制流边赋一个较大的权值
- 下次使用距离可以在活性分析上进行一定扩展维护得到

每个块需要决定初始时哪些变量在寄存器中(Wentry集合)

- 如果它不是循环头，那么尽可能使其前驱块保留的变量继续保留
- 如果它是循环头，尽可能保留在循环中使用到的变量，如果还有空余则保留穿过循环的变量

使用逆后序遍历来依次处理各个基本块

SSA上的寄存器分配

处理寄存器接合(Coalesce)

我们希望复制指令的两个变量尽可能位于同一寄存器中

`x <eax> = y <edx>  =>  x <eax> = y <eax>`

可以证明最优方案是NP完全的

启发式算法：

- 对于每条需要合并的指令，尝试调整染色方案使得变量处于同一寄存器中
- 如果调整成功，将这些变量对应的干涉图顶点合并

SSA上的寄存器分配

一些细节：

X86的一些指令要求操作数在指定的寄存器中，例如div和sal, sar等指令

无法保证要求的寄存器在指令执行时不被占用

可以在指令前加上一条平行复制指令，使得所有在寄存器中活动的变量的生命周期拆成两部分，强迫其重新分配寄存器，在此时可以指定寄存器的限制条件

(a’ <eax>, b’ <ebx>, c’ <ecx>) = (a <ecx>, b <ebx>, c <eax>)

d <eax> = a’ <eax> / b’ <ebx>

而这个平行复制指令可以在寄存器接合阶段被尽可能地优化掉

优化效果

测试数据	无优化	+寄存器分配	+内联	+无用代码消除	+全局值标号	+循环不变量优化
testcase_272.txt	0.935	0.341	0.102	0.108	0.088	0.092
testcase_274.txt	1.197	0.669	0.664	0.711	0.674	0.673
testcase_277.txt	1.101	0.571	0.267	0.268	0.268	0.253
testcase_279.txt	1.205	0.239	0.234	0.000	0.000	0.000
testcase_282.txt	11.034	2.414	1.504	1.517	1.112	1.039
testcase_284.txt	0.563	0.496	0.480	0.466	0.465	0.454
testcase_337.txt	0.479	0.104	0.046	0.046	0.033	0.022
testcase_338.txt	1.247	0.332	0.131	0.135	0.088	0.066
testcase_344.txt	4.603	1.066	0.727	0.724	0.675	0.623
testcase_348.txt	1.753	0.372	0.203	0.142	0.138	0.142
testcase_349.txt	4.956	0.654	0.506	0.506	0.470	0.479
testcase_350.txt	0.418	0.211	0.163	0.166	0.158	0.162
testcase_352.txt	0.540	0.167	0.080	0.097	0.084	0.089
testcase_353.txt	1.227	0.910	0.830	0.802	0.790	0.839
testcase_357.txt	1.311	0.331	0.163	0.162	0.154	0.141
testcase_360.txt	4.618	3.416	3.662	3.606	0.625	0.465
testcase_361.txt	34.233	8.043	3.235	3.202	1.935	1.619
testcase_362.txt	3.672	0.765	0.790	0.792	0.094	0.091
testcase_363.txt	3.915	0.915	0.566	0.543	0.392	0.398

参考资料

1. Engineering a Compiler
2. SSA Book: http://ssabook.gforge.inria.fr/latest/book.pdf
3. Hack S, Grund D, Goos G. Register allocation for programs in SSA-form[C]//International Conference on Compiler Construction. Springer Berlin Heidelberg, 2006: 247-262.
4. Braun M, Hack S. Register spilling and live-range splitting for SSA-form programs[C]//International Conference on Compiler Construction. Springer Berlin Heidelberg, 2009: 174-189.
5. Johnson R, Pearson D, Pingali K. The program structure tree: Computing control regions in linear time[C]//ACM SigPlan Notices. ACM, 1994, 29(6): 171-185.
6. Click C. Global code motion/global value numbering[C]//ACM SIGPLAN Notices. ACM, 1995, 30(6): 246-257.