

简单编译器

为了将程序的语法树转化为汇编代码主要需要经过这样一些环节。以处理 while+db 语言为例：

- 拆分复合表达式

由于汇编程序不能处理复合表达式，因此需要将复合表达式拆分成为一系列单步计算。在拆分过程中，需要引入额外的辅助变量。

- 生成基本块

汇编程序的结构不是树形结构，是一个汇编指令的列表，其中包含少量的跳转语句。因此需要将 AST 中的树形结构转化为这样以列表为主的结构。后面我们会详细介绍什么是基本块，什么是控制流图。

- 分配寄存器

汇编程序中没有变量，只有寄存器和内存。因此需要用寄存器或内存来存储程序变量（含先前阶段生成的辅助变量）。出于运行效率的考虑，我们应当尽量用寄存器来分配。

- 生成汇编代码

最后生成汇编代码是还可能补充实现一些细节，例如基本块的排布等等。

1 拆分复合表达式

- 关键点 1：每个赋值语句至多进行一步计算

- 关键点 2：转化后的程序仅在必要处保留树结构

- 关键点 3：要妥善处理 while 循环条件的计算语句

While 语句的处理方法

拆分前

```
while (old_condition) do {
    ...
    (old loop body)
    ...
}
```

拆分后

```

LABEL_1:
...
(some computation)
...
if (! new_condition) then jmp LABEL_2
...
(new loop body)
...
jmp LABEL1
LABEL_2:
...

```

- 关键点 4：要妥善处理短路求值

由于短路求值的存在，and 与 or 必须转化为 if 语句。例如：

拆分前

```

if (p && * p != 0)
then { ... }
else { ... }

```

拆分后

```

if (p)
then { #2 = * p;
      #1 = (#2 != 0) }
else { #1 = 0 };
if (#1)
then { ... }
else { ... }

```

2 生成基本块

3 寄存器分配

3.1 活性分析

- 计算方法：

$$in(u) = (out(u) \setminus def(u)) \cup use(u)$$

$$out(u) = \bigcup_{u \rightarrow v} in(v)$$

- 生成 interference graph：如果存在一个 u 使得 $x, y \in in(u)$ ，那么就在 x 与 y 之间连一条边，表示它们不能分配同一个寄存器。

3.2 简单寄存器分配算法

本课程只考虑最简单的寄存器分配，即每个变量对应一个寄存器，多个变量可能对应相同的寄存器。较为前沿高效的寄存器分配算法，有时会对同一寄存器的不同使用位置分配不同的寄存器。

- 假设共有 K 个寄存器；

- 步骤一, Simplify: 在 interference graph 中删除度数不超过 $K - 1$ 的节点, 删除的节点用栈记录;
- 步骤二, Spill: 如果无法 Simplify, 就任意删除一个节点, 再回到前面步骤一;
- 步骤三, Select: 按照删除节点的倒序为所有变量分配寄存器;
注: Simplify 删除的节点能够保证被分配到与 interference graph 上相邻节点不冲突的寄存器, Spill 变量可能可以分配到寄存器 (假 Spill), 也可能无法分配到寄存器 (真 Spill), 此时应当存储在内存中;
- 步骤四, Start over: 如果有至少一个节点无法分配到寄存器, 则改写相关代码并重做 liveness 分析与上述所有步骤。
- 如果 3 号变量无法分配寄存器, 并且存储在 `%rbp - 16` 地址, 那么:

```
#2 = #3 + 1
```

```
#3 = * #4
```

分别会被改写为:

```
#3 = * (%rbp - 16)
#2 = #3 + 1
```

```
#3 = * #4
* (%rbp - 16) = #3
```

这样, 两次使用 `#3` 变量之间的代码 `#3` 都不再是 live 的了。

- 如果所有变量都分配到了寄存器, 那么寄存器分配过程结束; 如果有至少一个节点无法分配到寄存器, 那么在执行完上述程序改写后, 变量的 live 区域会大大缩小, 此时应当重新进行 liveness 分析, 并回到步骤一, 从头开始重新分配寄存器, 这个过程称为 Start over。往往经过至多 2-3 次 Start over 之后, 寄存器分配算法就能顺利结束。