如何设计三地址中间代码的数据结构，以便于基本块分析和代码优化？ - 知乎

https://www.zhihu.com/question/33518780/answer/56731699

RednaxelaFX计算机科学等 7 个话题下的优秀答主 Sun Aug, 08 23:20

问题背景：题主正在做的编译器习作的源语言的C的子集（某种C-Minus？），而实现语言是Java。
并且IR的设计是想用龙书系的三地址指令（TAC，three-address code）来做。

龙书讲前端的部分感觉还比较直观，而且有第二章以及附录A的实际代码例子，就算是初学者也应该比较好着手学习；但是讲后端的部分由于没多少配合演示的代码，确实容易感到怎么读都似懂非懂，然后又不知道应该如何组织实际代码去实现书上讲的知识点。
对此我自己深有体会，头几次读龙书都觉得后端的部分似乎读懂了但还是不知道该怎么下手写代码。所以很乐意来回答题主的问题，好让后人少走弯路。

而《Crafting a Compiler》（编译器构造）我觉得写得详细是挺详细，就是有点浅所以读得很不过瘾；而且在前端花了太多篇幅，有很严重的虎头蛇尾感。
不过这个配置或许正好符合原本作者的目标读者群的需求吧。它前面讲得详细的部分真的很详细，可操作性很强，倒是挺适合初学用。

==========

内存还是文件？

现代编译器的（主要）都是在内存里一口气完成整个编译流程，而不向外写中间文件的。所以题主可以主要考虑in-memory形式的IR。

例外之一是微软的Visual C++编译器。它继承了以前内存极其紧张时的分阶段式设计，将编译器前端（c1.dll/c1xx.dll）与编译器后端（c2.dll）分开在两个不同的进程里，两者由一个驱动进程（cl.exe）协调；前端用中间临时文件来传递AST/IR给后端。这个设计一直延续了下来…
像PCC这种老爷编译器也是前后端分离，中间用临时文件传递IR。

如果想易于开发/调试的话，配套设计一种文本形式的IR也是极好的。LLVM IR就是这方面一个非常好的例子。这样会很利于单独测试编译器里的某些IR变换。
受LLVM IR启发的MicroVM的Mu IR也设计为有in-memory形式和配套文本形式。其中一些IR指令的in-memory版的Scala实现可以看这里：https://github.com/microvm/microvm-refimpl2/blob/master/src/main/scala/uvm/ssavariables/ssavariables.scala

> 就是不知道下面的三地址形式的中间代码不知道如何存放，是直接放到一个文件还是放在内存中的数据结构

所以回答这个问题：请先考虑以in-memory形式为主，这个是主干；然后可选的，作为利于调试的手段配套设计一种文本形式；两种形式之间最好能做到无损的相互转换。
==========

IR设计

然后具体要怎么设计IR便于优化…这个水就深了，而且虽然有些选择非常重要，更多的buzzword选择其实并不真的很影响功能，而更多的是个人代码风格的问题。就像大括号写行末还是换行写还是换行+缩进写一样…
我不想在“个人偏好”方面乱放炮，总之题主自己摸索一下选择自己喜欢的就好 >_<

下面针对龙书系的三地址IR来展开。以虎书为代表的树形IR就按下不表了。
TAC的典型实现方式是Quad（四元组），一个operator加上3个operand。其典型形式是：

或者有些记法把它记为：
两种记法指的是同一种东西。
四元组，顾名思义，其实就是一个数据结构把四个字段打包起来：

不过实际写代码时这里有许多东西要选择：

- 是否需要显式的“变量”结构？就是说，让IR分为两种结构，一种专门表示变量，一种专门表示“运算”，每个运算结果一定要赋予一个（或多个）变量；还是用一种统一的结构，既表示运算又表示该运算的值。在做高层优化时，使用统一结构比较易于操作；而在后端做寄存器分配的时候，有显式的“变量”概念利于记录寄存器分配信息。有些编译器会采用多层IR，在上层IR不带显式“变量”结构，而在底层IR带有显式“变量”结构。“变量”结构在实际代码里可能叫作Var、Operand、Opnd之类的名字。
- 如果不使用显式的“变量”，那么operands是直接引用其它Quad IR的指令？还是用某种压缩的形式？例如说把所有IR指令都放在一个大数组里，那么operand就可以用数组下标来表示。通常链式结构更易于操纵，但压缩形式更省内存。习作的话我还是推荐用链式结构。采用下标来表示operand最麻烦的就是假如IR数组代表了执行顺序，而我们要在某个IR指令前面插入几条新的IR指令的话，后面的IR的下标就全都要重新计算一遍并且更新，特别麻烦——当然也有应对的解决办法…
- IR指令是否需要串起来？要的话，用数组（或例如ArrayList）还是链表？链表的话用单向链表还是用双向链？这里也是，通常链式结构更容易操纵，我会推荐先用链式结构来实现。
- 控制流，是在IR里设计一种LabelNode来表达跳转目标，还是使用显式的CFG（基本块构成的控制流图）+Quad，还是把控制依赖跟数据依赖融合到同一种表现形式（PDG）？传统做法是CFG+Quad，龙书默认大家都是这么做的。

- Quad是用传统的非SSA形式，还是用SSA形式?
  - 如果用SSA形式的话，是只对local variable（也叫scalar variable）做SSA，还是对内存也做SSA（memory-SSA或者heap-SSA）？
  - 如果用传统形式的话，是否或者如何维护use-def关系？是在辅助数据结构里维护ud链（use-def chain），还是fud链（factored use-def chain），等等?
  - 我的习惯是直接进SSA形式，然后在IR里一直保持SSA形式一直到寄存器分配做完才退到传统形式，但也有很多人正好相反，几乎全程都用传统形式，只在中间的特定优化阶段转入SSA形式。
- …还有很多

别着急，上面这些问题都不是“致命选择”。条条道路通罗马，虽然有些路走起来会有点崎岖…题主不要害怕，一开始每个地方都随便选一种做法就好了。后面写得多了就会慢慢体会到某些选择或许选别的路更好，重要的是最初要迈出第一步做出第一个选择。
采用最最最传统的方式来做的话，可能会写成：

```
enum Opcode {
  // add, sub, mul, div, load, store, if, goto, call, return, label ...
}

// 变量
class Var {
  // ...
}

// 一条IR指令
class Quad {
  // 四元组
  Opcode op; // 单独把opcode写做一个字段挺不Java的，许多C/C++写的编译器喜欢这样，Java可以用继承
  Var src1;
  Var src2;
  Var dest;

  Quad prev, next; // 双向链
}

// 整体IR
class IR {
  Quad head; // 用一个线性链表来表示整个函数的逻辑；没有显式CFG，用label来指定跳转目标
}
```

这样用单一结构体（例中Quad类）作为IR的指令的做法常见于用C写的编译器。Java的话可以对IR类型创建一套类层次，通过不同的子类来表达不同opcode的语义；这些子类也不需要采用同样的layout——有些子类可以只要更少的字段（例如实现单目运算符就只需要1个src operand），而有些可能需要更多字段。

龙书的第二章和附录A所演示的前端范例生成的中间代码是一种用label表达控制流的线性代码，没有显式构造CFG。这种代码可以很直观的映射到上面的IR设计。
但是后续要做优化需要控制流信息时，还是得遍历一次这个线性IR去显式构造CFG，我觉得反而麻烦…

如果是要做一个非常非常简单的编译器，其middle-end到back-end只有一种IR贯穿其中的话，那采用这种IR也有一定道理：它在从AST翻译过来的时候不需要显式创建CFG，而最终要生成目标机器代码时对应关系比较好——目标机器也多半不会有显式的CFG概念，而是用label+有fallthrough语义的branch来实现条件控制流，正好跟这种IR一样。这就可以让中间需要CFG的优化变得完全是可选的，而不必一开始就非得实现CFG不可。
但是，一旦开始想认真做更多优化，这种IR就会让人感到特别蛋疼…

然后假如不要变量、要CFG、要SSA的话，可能会写成：

```
enum Opcode {
  // add, sub, mul, div, load, store, if, goto, call, return, phi ...
}

// 一条IR指令
class Quad {
  // 四元组
  Opcode op;
  Quad   src1; // 直接使用别的指令作为operand
  Quad   src2;
  // 不需要指定dest; 自身就代表了运算和值

  Quad prev, next; // 双向链
}

// 一个基本块
class BasicBlock {
  Quad head; // 用一个线性链表来表示整个基本块的逻辑; 末尾必须是控制流指令; 中间不包括phi
  ArrayList<Quad> phis; // 记录该基本块开头所需的phi指令

  // ArrayList<BasicBlock> successors;
  // 后继基本块; 该信息可以从该基本块末尾的控制流指令提取, 不需要显式用字段记录

  // ??? flags;
  // 可选的额外信息, 可以用来记录诸如按reverse post-order (RPO) 遍历CFG的序号, 是否是循环开头 (loop header), 等等
}

class IR {
  BasicBlock start; // 起始基本块

  // List<BasicBlock> blocks; // 所有基本块
  // 虽然从start肯定能顺着CFG遍历到所有活着的基本块, 但也有些实现为方便而在一个列表里记录下所有基本块的引用;
  // 有时候 (例如要处理异常的话) 有可能有活着的基本块会无法直接从start到达, 此时也最好显式记录所有block。
}
```

是不是有点区别?

许多编译器都会选择直接从AST直接生成带CFG的IR。要参考例子的话，LLVM Tutorial的Kaleidoscope是个好去处。

我写这段的时候LLVM官网正好挂了，所以放个镜像网站的链接：[LLVM Tutorial: Control Flow]

IR的形式显式的信息越多（例如有显式CFG，采用SSA形式之类），在最初构造这种IR时所需的处理就越多，但是相应的后面做分析的时候就越方便，特别是如果要做稀疏分析（sparse analysis）的话。
这些记录在IR里的额外信息很多都可以看作是一种“缓存”——即便不记录在IR里也可以在后续分析需要的时候从头遍历一次IR来计算出来。但是缓存起来就不用每次重新算，所以比较方便——然而缺点也很明显：这些信息存在IR里是得维护的，有维护成本；每次IR发生变化，缓存的信息可能都得跟着更新。

就看IR的设计者想如何取舍了…

==========

实际项目的例子

龙书系的Quad设计，一种实际实现是JoeQ里的IR。
请参考Stanford CS243课程的资料：http://suif.stanford.edu/~courses/cs243/joeq/#quads
现在可用的一份JoeQ的代码可以从这里下载：http://suif.stanford.edu/~courses/cs243/hw2/hw2.zip，其中lib/joeq.jar里既包含程序的Class文件也包含源码。
JoeQ是一个用Java实现的“元循环JVM”，内含一个用Java写的、能把Java字节码编译为机器码的编译器。
JoeQ IR就是这个编译器里的IR，实现形式为显式CFG+Quad。Quad可以是传统形式也可以是SSA形式。

@啥玩应啊 的回答提醒了我，Soot的Jimple IR也是TAC。有很多静态代码分析工具基于Soot框架实现，甚至也有直接把Soot用在编译器里作为前端的（例如RoboVM）。Jimple IR的设计也可以参考，源码在Soot的Github站上可以找到。

我比较熟悉和习惯的一种IR是HotSpot Client Compiler（C1）的HIR。它原本用C++实现，不过有个Java移植版C1X，可供题主参考。C1X的HIR实现源码在此：https://kenai.com/projects/maxine/sources/maxine/show/com.oracle.max.c1x/src/com/sun/c1x/ir?rev=8809
C1 / C1X的一些设计选择是：

- 在HIR里不使用显式变量，每条IR指令既代表一个运算也代表该运算的值
- 每条HIR指令的operand直接引用其它HIR指令
- HIR指令通过单向链串起来
- 有显式的CFG，HIR指令都挂靠在其所属的基本块（用BlockBegin指令表示）中
- HIR使用SSA形式，不过只对local variable做SSA而不对内存做SSA。结合前面提到的直接引用其它HIR指令作为operand，从数据依赖的角度看，每条IR指令带有完整的数据依赖描述——每个use-site都可以直接找到其对应的def-site。这种设计的概述请参考Design of the Java

HotSpot™ Client Compiler for Java 6论文。也有地方把C1/C1X的HIR形式叫做VDG（Value-Dependence Graph）：C1X | Oracle Community

> C1X retains most of C1's original design, with some front-end improvements, simplifications, and cleanups while porting to Java. The C1X frontend parses bytecodes into an SSA-style value-dependency graph that retains the original basic block structure of bytecodes. Because of the value-dependence graph nature of this main IR, operations on local variables and the stack result in no IR nodes being generated (unlike a named-form SSA or three-address form of IR). This representation is remarkably convenient for forward-dataflow problems such as constant-folding, value-numbering, and null-check elimination.

值得提醒的是，虽然上面的文档把C1/C1X HIR叫做VDG，它这么叫只是为了突出“一个值直接依赖于别的值，而没有变量结构”的特点，而并不是指那种专门叫做VDG的IR形式：https://courses.cs.washington.edu/courses/cse501/06wi/reading/weise-popl94.pdf

我比较熟悉、习惯和喜欢的另一种IR是HotSpot Server Compiler（C2）的Sea-of-nodes IR。这是个PDG（Program Dependence Graph）的变种，在编译器中很非主流。PDG更多见于非编译器的静态分析器中，所以就不在这帖多说…

但是为了安利一下Sea-of-nodes形式，还是放个Cliff Click大神读PhD时写的论文的传送门好了：From Quads to Graphs: An Intermediate Representation's Journey (1993)。

等题主在Quad里摸爬滚打足够时间后再读这篇论文可能会有更多体会 >_<