Report for Compiler 2015

李文昊
5122409038
F1324004

2015.5.30

Abstract

这篇报告整理了我用 java 实现简易版的 C 语言编译器时的相关工作，其中包括 CST、AST 的建立极其相关工具 Antlr 的大概使用，语义检查，生成中间代码，生成最终代码与一些细节。

Contents

1 Introduction

本课程的最终目标是让我们实现一个能把简易的 C 语言代码（具体要求在课程主页上）翻译成 MIPS 码的编译器，根据助教给我们的任务阶段划分与我自己的实际情况，大概分为四个阶段。

1. Syntactic Analysis: 这个部分大概是要对输入的文本形式的 C 语言代码进行基础的语义分析，其中包括词法分析（lexing）和句法分析（parsing），词法分析大概是把这些看似无关的连续字符给划分成许多个 token，从而得到最小的语义块，而 parsing 就是根据一个给定的"规则"把这些 token 组合出来的语法结构给抽取出来。在这个块，我虽然尝试了自己写一个 lexer，但是最终还是使用了工具 antlr。

2. Semantic Check: 在构建好 AST 后，虽然已经得到了整个结构，但是有些结构可能在具体语义上并不合理，比如我们并不能把一个 struct 类型的变量赋值给一个 int 类型的变量，也不能把两个 struct 类型的变量相加。基于这些考虑我们需要在这一步把这些不合理的现象给剔除。由于我建立了 AST，所以在 AST 遍历的同时，对这些情况进行判断，从而完成这个工作。

3. Intermediate Representation：我们的最终目标应该是生成 MIPS 码，但是直接这么转换并不好，因为在 MIPS 码中，我们还要考虑寄存器的分配，和各个全局变量，局部变量的存储位置等复杂的因素，而这些因素将会成为我们要优化的重中之重，如果在这个部分就"一步到位"将会导致我们后面的 debug 异常艰难，修改也会变得十分难受（牵一发而动全身）。所以为了避免这种问题，我们可以先生成一种类似"MIPS"码的中间代码，先不考虑寄存器等问题，把代码中的基本信息先用一种简单的语句 – 四元式的形式表示出来。

4. Code Generation：剩下的最后一步就是要通过中间代码来生成最后的代码了。因为中间代码足够简单的缘故，所以我们可以把注意力更多的放在如何分配内存，取实际的地址上，从而生成最终的 MIPS 码

以下我会对这些部分加以更加详细的说明。

2 Syntactic Analysis

由于一开始对整个语义部分并不理解，导致我十分迷茫的看了很久龙书，但事实表明这个并没有什么用，龙书上虽然讲的很清晰易懂，但对于这个阶段来说还仅仅是理论上的东西，而要开始这项工程还需要更多的实践。

处于对这一块的不理解，我最终选择了用 java，并且使用 Antlr 来生成我需要的 CST。使用 Antlr 在这个阶段着实是一个很大的难题，因为 Antlr 官方

提供的资料是一本书，而不是一个文档，所以内容十分零散，要查起来也十分费劲，常常产生看了很久都没有看到自己正需要的内容的情况。

于是为了摆脱这个情况，我在阅读了 Antlr 前 4 章的内容，知道了一些基本命令并用助教给的文法要求生成 CST 后，我就开始通过询问廖超学长，并且查看一些早期学长的 AST 实现来学习如何通过在 Antlr 的 g4 文件中添加 java 代码来构建 AST。

也就是在这个过程中，我对 AST 和 CST 的理解又加深了不少。我认为 CST 由于直接由语法构建出来，在很多细节上会显得不可避免的繁琐，比如对于一个表达式，为了严格表现其运算优先级，就算是对于再简单的 a = b + c 这样的算式，我们也要从逗号表达式来生成。这样一来，CST 上就会有很多并不需要的节点。而 AST 就十分高效的处理掉了这些情况，不仅仅删除掉了无用的节点，还把一些必要信息进行了上提，比如对于 a + b，我们把加号提到了树的内部节点上，而不再仅仅作为一个叶子信息保存在树的最下端。这样一来更加符合人们对这些语法的抽象理解，极大的方便了后续的工作

如果用的是 java，并且需要把 CST 转换为 AST，必须要先设计好一个整体的框架，这个框架必须做到，保留足够的信息，并且尽量的简洁高效。还好 java 是一个面向对象的语言，我们可以通过抽象类来编织一个优美的 AST 框架。由于怕出错，我最后还是使用了廖超的框架，事实也证明，这个框架如果后期不加一些很高级的特性，也不需要修改。

在把 CST 转换成 AST 的过程中，应该要想明白在 CST 上 dfs 的“顺序”。我使用的是廖超介绍的用全局变量来提取出关键信息的方法。而这种方法就更加需要理清楚 dfs 时顺序的思路，因为修改如果修改的是全局变量，必须要想清楚这样一来是不是会对其他部分产生影响。

最后为了显示出 AST 构建的结果，我为每一个类写了一个 draw() 的方法，并且通过缩进让这些看起来更加的美观整洁。

3 Semantic Check

语义检查这一部分大概可以分成两个部分，第一个是符号表上的问题，一个是一般的语义上的问题，包括 Declaration，Statement 与 Expression 这三个部分的问题。

1. 对于一个程序，如果在全局有一个 int a; 的定义，而在某个函数内又有一个 int a; 的定义，那么如果我们在函数里面对 a 进行了修改，那么到底是哪个 a 的值产生了变化呢?

对于这个问题，修改会发生在往上层走最近定义的一个 a。如果在全局定义了两次 int a = 0; 同样也是不合法的。那么这就需要我们为此专门写一个“符号表”来检查出这些问题。我为此专门写了一个叫做 Environment 的类，里面用 table 存储了符号表，符号表具体存储了到达 AST 的某个节点时，它能访问到的变量，函数以及类型名称。由于在进入，脱离某个命名空间的时候，需要对在这个命名空间中加入的定义进行“撤销”操作。对此我参照了虎书的第五章，维护一个 hash table，并且通过链表的方式来支持撤销。为了防止变量名重名问题，还在有全局时定义变量的种种特殊性（比如可以重复定义 int a; 而不会变量名定义重复），我对每个符号都记录了他的层数（Level）。

2. 对于 declaration、statement，主要就是一些 break，continue 等判断，但是有一个比较难处理的东西就是初始化列表。由于 C 语言的初始化列表十分复杂，有太多太多的特例，在最后的语义检查的测试阶段也有所体现。我为此做了很多特殊判断，个人感觉这部分实现不是很优美。而对于一个 expression，我记录了三个值，一个是类型，一个是它是否是左值，还有一个就是他是不是常量了，如果是，我还会把这个具体的值给记录下来。这样一来在一些情况下可以减少一些可以预见结果的计算。在这个部分我觉得一个比较大的难点是类型转换，如何判断在不同条件下两个类型是否可以相互转换。还有一个难点就是在二元运算中的庞大的分情况考虑。

不过总体来讲，这个部分还是比较轻松地，有数据可以及时的测试，而且查起来也很方便，和后面的 Code generation 的 debug 比起来真是好太多。

4 Intermediate Representation

这个部分我还是沿用了助教给出的架构。由于并没有理解助教的架构的具体意义，导致在生成的过程中遇到了巨大的阻力，最后也是发现错的太多而不得不重写了好几遍，在这个过程中也对助教的架构进行了许多修改，虽然最后表明这些修改都是没有必要的。

在我的架构上，在比较高的层面上可分为四个部分，分别是 Address，Variable，Quadruple 以及 Function 这四个部分。其中 Variable 记录了变量名称以及相对应的变量信息，比如它的大小以及类型。而 Function 这个部分就是变量定义和 Quadruple 的集合体。而这四个之中的重中之重我认为是 Address。

Address. 在廖超的架构中，Address 大概分为一个在程序中实际定义出来的变量 (Name)、中间计算过程中需要的临时变量，一些常量。但在我后面写代码生成时，发现其实 temp 和普通的 Name 是一个东西，都需要存在栈中，所以我把 temp 这种类型直接转移成了 Name，并且删除了 temp 这种类型。

Variable. 记录好这个变量的大小即可。

Quadruple. 需要理解好架构中各个 Quadruple 的作用，这样才能更好地分拆原代码中各个复杂语句。

Function. 存储好在这个 function 中定义的变量，而且要把定义的临时变量也计算入内，这样一来可以在 Code generation 的 functionCall 时就能静态算出要开多大的栈帧。

虽然这个阶段独立出来作为一个 Phase，但这个部分的最后是没有任何测试数据的，所以并不显得独立。在实现上，和 semantic check 十分类似，也需要维护出一个符号表，对于每一个 expression，我也需要提取出三个关键信息，分别是它的 type，address，以及是否为常量。

这里的一个难点在于和 code generation 有很大的关连，而且如果事先对 MIPS 没有一定了解，根本不知道怎么写是对的，很可能在写 code generation 的时候才发现架构上的信息并不完整而需要对架构进行大改，而我在这次也是重写了三遍才对... 如果让我再写一次编译器，我应该会自己独立设计一个架构，使得其和后面的 code generation 更加契合

5 Code Generation

最后的部分就是生成 MIPS 代码了，在上一步生成的 IR 架构上进行遍历，把现在相对简单的四元式翻译成 MIPS 码。这个的难点在于地址的分配。

由于我在上一步并没有对变量进行重新标号，但是由于我对每一个变量（包括临时变量）都新建了一个类 Name，而我在 IR 中遍历时，遇到一个变量，对他的索引将不再是字符串的名字，而是一个类（即 Name），这样可以直接处理掉变量名字冲突的问题。

并且对于每一个不同的 Name，我会记录他的准确的地址，如果是全局变量，我会记录好他的名字，并且每次 la 出具体地址。如果是局部变量，我会记录他的相对栈指针 $sp 的相对位移 offset。

由于这部分总体写下来其实比较简单，但是细节比较多，需要自己好好理清楚，如果思路清晰，正确性一个下午就能通过，否则就要好几天的无休止调试。接下来是我调试中出现的一些问题。

1. 对于很多变量，可能是 char，也可能是 int，这两个在 lw(lb) 或者 sw(sb) 上会有所区别，那么如何判断到底是 4 字节还是 1 字节其实是一个很蛋疼的问题。我为此专门写了一个 Map，来记录每一个 Name 的具体大小，并且对 load 和 store 进行了一定程度上的封装，可以极大地简化代码，并且避免很多重复性

带来的错误。

2. 最容易考虑不清楚的地方其实函数参数传递，对于 struct 或者 union 类型，是要进行值复制而传递，而对于数组，可能只是传递一个地址过去。而且要理清楚到底传递过去的值是一个地址，还是说是一个准确的值。我也就是在这里浪费了挺多时间。

3. 手写了 printf 的 MIPS 码，由于 printf 是不定参数的函数，所以在写法上会有点特别。刚开始不熟悉 MIPS 码可能会有点不适从，不过多尝试几遍之后还是比较容易手写的。手写 printf 代码可以大幅减少指令数

关于优化，我只实现了一点基本的窥孔优化，比如在 IR 中进行了一些特判，把一些明显的死代码进行了消除，然后把一些常量直接传递，而不用到临时变量。把一些没必要的语句都进行了合并。

6 debug

这个部分是我在 code generation 中花费的最多的部分。因为 spim 检验代码时，就算错了，很难知道是哪部分错误。大部分时候必须要从 IR 开始查。但是在这个过程中我也是慢慢掌握了自己的一套调试方法，以下是我的一些心得。

1. 通过二分的方法找到错误的位置，把多余的代码或者通过简单的赋值替换，或者通过直接删除，来达到简化代码的作用，目的是使错误的地方就只有一个尽量简单的语句。

2. 观察 IR 生成的中间代码，检查是否在这个阶段就出现了问题。如果是就要更改 IR 的代码

3. 观察 MIPS 代码，由于我使用的是三寄存器法，所以查起来相对容易一些。

4. 如果能确定自己 printf 是正确的，那么可以通过在源程序中加入 printf 来输出中间结果发现问题。

5. 多思考，而不要一味的看代码。有时候一个错误反应的可能是一个之前设计上的大问题。不要只改了一个小错误而没注意到大错误。

7 gain

又一遍熟悉了 git 的使用方式，对其的版本控制功能得以一窥。

第一次写了一个这么大的工程，对于工程有了初步的认识。

第一次写了这么长的 java 程序，对于 java 的很多特性也用的更加熟练了，对以后的工程实现应该也会有很大的帮助。

对于计算机如何解析代码有了更加深入的认识。

对于庞大的工程，一定要随时整理自己的思路，多想想好过乱写一通最后删掉。但是在某些时候，总是想而不动手可能根本不知道会发生什么问题。所以这两者之间要找到一个折中方式

...

8 thought

不得不说，写了这么大一个工程，虽然最后并没有写什么 bonus，但是还是很有成就感的。虽然其中大部分都是一些机械的学习和码代码，但是最后还是收获了很多之前没有的经历。

9 Reference

[1] *Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. Compilers: Principles, Techniques, and Tools. China Machine Press, 2009.*

[2] 《*Compilers Principles Techniques and Tools*》

[3] *Kai Sun, Shunning Jiang, Chao Liao, Shuang Liu, and Wen Xu. Compiler 2015.*

[4] 《*Modern Compiler Implementation in Java*》

[5] 《*James R. Larus. Assemblers, Linkers, and the SPIM Simulator*》