LoliCCompiler

LOLI : LOLI Oriented Language Implementation

倪昊斌/RobbinNi

2015.4~5

~核融合炉 Ver~

目录

1. Introduction

LoliCCompiler 是上海交通大学 13 级 ACM 班编译原理课程的一项学生项目。作者为 13 级 ACM 学生倪昊斌。目标实现一个采用现代编译器的通用架构模式，以 C 语言的一个较为精简的子集为源语言，以 MIPS 指令集的汇编器 SPIM 为目标机，并能够进行一些较为基本的编译器优化的编译器。编程语言是 JAVA。本文就这一项目所采取的基本架构、所支持的特性和部分具体实现进行报告式的介绍。

1.1 基本架构

LoliCCompiler 整体可分为五个部分：

1、分词和语法分析(Lexing & Parsing)

- 原程序(代码)→符号流→抽象语法树(AST)
- 检查原程序中存在的语法错误

2、语义检查(Semantic Check)

- 抽象语法树(AST)→中间表示树(IRT)
- 检查原程序中存在的语义错误

3、中间表示(Intermediate Representation)

- 中间表示树(IRT)→三地址代码(IR)
- 静态单赋值形式(SSA)
- 控制流优化

4、代码生成(Code Generation)

- 三地址代码(IR)→MIPS 汇编代码
- 全局（跨过程）寄存器分配

5、优化(Optimization)

- 公共子表达式消除
- 冗余复制消除
- 死代码消除

而这一架构也是现代编译器多采用的围绕不同的中间表示之间的转换进行的模块化架构的一个微缩版本。后文将按照这一顺序逐模块进行介绍。

1.2 主要特性

LoliCCompiler 支持许多独特特性，具体可分为语言特性、功能特性和优化特性三类。

1.2.1 语言特性

1、复杂类型系统

• 支持结构/联合类型、函数类型、数组类型、指针类型的任意嵌套。

2、内嵌函数支持

• 可以在函数体内定义仅能在这一函数内调用或通过函数指针进行调用的内嵌函数。

3、未定大小数组

• 允许声明未定大小的数组类型。LoliCComplier 将会根据初始化列表大小等信息自动计算数组大小。

4、typedef 支持

• 允许通过 typedef 简化复杂类型的定义。LoliCCompiler 可以正确区分被 typedef 的类型和与之同名的变量。

5、函数指针

• 可以定义、赋值、传递、调用函数指针，以实现多态、代码隐藏等功能。

6、高阶函数

• 对于内嵌函数，支持通过函数指针将其作为参数传递。LoliCCompiler 实现了 gcc 的 trampoline 技术来支持对于调用函数的变量的正确访问。但执行运行时生成代码这一特性目前暂不被目标机 SPIM 所支持。

7、scanf 支持

除了 printf/getchar/malloc，LoliCCompiler 和 LoliCInterpreter 均额外支持 STL 中的 scanf 函数，支持%d、%s、%c 作为描述符。

1.2.2 功能特性

1、用户友好型 GUI

• 设计简洁的 GUI 控制面板集成了 LoliCCompiler 的各项特性。用户可以方便而清晰地看到 LoliCCompiler 的内部中间表示细节。

2、代码美化工具

• 可以对于一段源代码进行自动缩进、插入空格来进行美化，使之更易于阅读。

3、语法/语义错误信息

• 对于语法和语义错误，能够在 GUI 界面上返回具体的错误信息。

4、C 语言解释器(LoliCInterpreter)

• LoliCCompiler 内含一个功能完整的 C 语言解释器(LoliCInterpreter)，可以直接运行 C 语言源代码，并且实现了与用户进行 IO 交互的仿控制台。

1.2.3 优化特性

1、控制流优化

• 在代码生成中，对于分支语句判断表达式的计算进行了分析，能够大量减少生成的跳转指令。

2、静态单赋值形式(SSA)

• 将中间表示转换为静态单赋值形式(SSA)并进行优化再转回三地址代码进行代码生成。

3、全局寄存器分配

• 正确实现了循环分析、过程间数据流/控制流分析、符号寄存器汇编生成、并以此为

基础实现了溢出改进的图染色寄存器分配算法。优化效果显著。

4、公共子表达式消除

- 能够提出在程序中出现的公共子表达式，并通过替换减少计算量。

5、冗余复制消除

- 基于 SSA，能够通过等价替换消除冗余的复制指令。

6、死代码消除

- 基于 SSA，对于定义后不再会被使用的变量，其定义会被消除。

2 Lexing & Parsing

分词和语法分析(Lexing & Parsing)是 LoliCCompiler 的第一个模块。主要是将源代码通过分词转化为符号流，再通过语法分析转化为抽象语法树。我使用了正则表达式+Jflex 生成分词程序，上下文无关文法+JCup 生成语法分析程序，并通过在语法分析中插入 action code 来实现抽象语法树(AST)的构建。

2.1 分词：正则表达式+Jflex

分词完成由源程序(字符串)到符号流的转化。一个符号(Token)是语言中表达最基本含义的单元。一个符号可以用一个正则表达式来表述，字符串和正则表达式的匹配则可以由确定状态自动机高效完成。对于在程序当中某一处有多个符号可以匹配的情况采用最长优先原则。

我通过描述正则表达式、设定不同的分词状态并加入 Java 代码处理转义字符，使用 Jflex 工具生成了分词代码。

```
118  commonCharacter = [[\x20-\xff]--['\"\\]]
119  translatedCharacter = \\[bfnrt\\'\"0]
120  asciiNumberCharacter = (\\x[0-9A-Fa-f][0-9A-Fa-f])|(\\[0-3][0-7][0-7])
121  //char = {commonCharacter} | {translatedCharacter} | {asciiNumberCharacter}
122
123  %state YYSTRING
124  %state YYCOMMENT
125
126  %%
127
128  <YYINITIAL> {
129      /* Pre-Compile Command */
130      {preCompileCommand}       { /* ignore */ }
131
132      /* Comments */
133      {singleLineComments}      { /* ignore */ }
134      "/*"            { yybegin(YYCOMMENT); }
135
136      /* Keywords */
137      "void"          { return symbol(VOID); }
138      "char"          { return symbol(CHAR); }
139      "int"           { return symbol(INT); }
```

图 1：loliccompiler.flex 片段

2.2 语法分析：上下文无关文法+JCup

分词产生的符号流输入到语法分析器中。语法分析器的作用是识别语法结构，并根据所识别的语法结构建立抽象语法树。这里也可以通过递归下降法来手写分析。但是为了支持一些较为复杂的语法模式和 typedef 所要采取的 lexer-hack，我使用了上下文无关文法来描述 C 语言的语法再 JCup 自动生成语法分析器代码的方法。通过在识别出特定语法模式时执行相应的 action code，可以建出抽象语法树。

```
224  declarator ::=      direct_declarator:dect
225                  {:  RESULT = dect;  :}
226                  |   MUL declarator:dect
227                  {:
228                      if (dect.type == null) {
229                          dect.type = new PointerType(null);
230                      } else {
231                          dect.type = dect.type.encore(new PointerType(null));
232                      }
233                      RESULT = dect;  :}
234                  ;
235
236  direct_declarator ::=   IDENTIFIER:name
237                      {:  RESULT = new VariableDecl(null, new Symbol(name), null); :}
238                      |   TYPENAME:name
239                      {:  table.addEntry(name, Symbols.IDENTIFIER);
240                          RESULT = new VariableDecl(null, new Symbol(name), null); :}
241                      |   PARAL declarator:dect PARAR
242                      {:  RESULT = dect; :}
243                      |   direct_declarator:dect PARAL {: table.addScope(); table.delScope(); :} PARAR
244                      {:
245                          if (dect.type == null) {
246                              RESULT = new FunctionDecl(new FunctionType(null, new DeclList()), dect.name,
247                          } else {
248                              dect.type = dect.type.encore(new FunctionType(null, new DeclList()));
249                              RESULT = dect;
250                          }
251                          :}
252                      |   direct_declarator:dect PARAL {: table.addScope(); :} parameter_list:para {: table
```

图 2：loliccompiler.cup 片段

2.3 抽象语法树设计

抽象语法树是 LoliCCompiler 中第一个对于源代码的等价表述。其主要目的是描述程序的抽象语法框架。我的抽象语法树设计分为以下 3 个主要模块和 2 个辅助模块：

declaration：函数、变量等的声明。9 个类。

statement：循环、分支、跳转语句。10 个类。

expression：表达式树。16 个类。

type：类型系统。12 个类。

initialization：初始化列表。3 个类。

它们之间的关系可以用这样一张图来描述。Program 是整棵 AST 的根。箭头表示了类型之间的为另一类型的属性的关系。

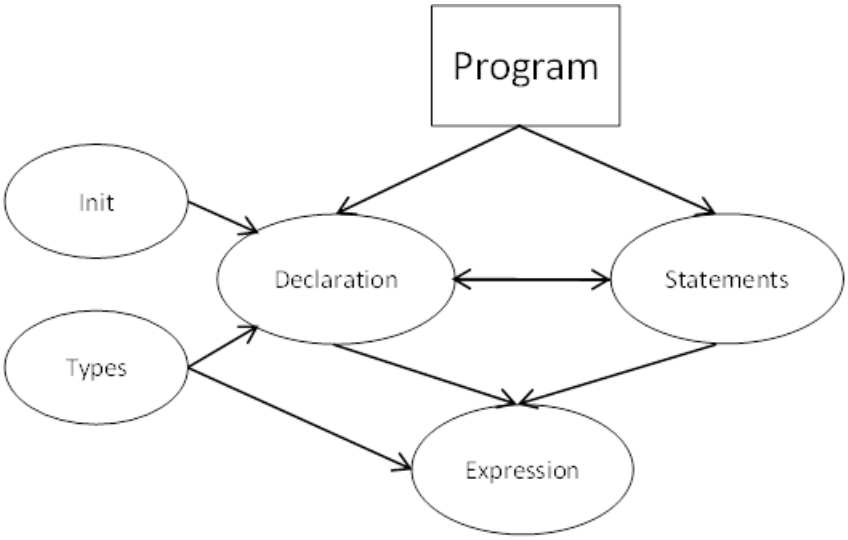

图 3：AST 类结构图

在遍历 AST 时我大量采用了访问者模式。通过不断地把控制权在具体类和访问者之前转换实现对于 AST 的遍历。这样做的好处是面向对象的 AST 的每个类可以只保留自己的属性定义，而把遍历的代码通过访问者集中起来进行管理共享环境。不过在遍历时传递参数会受到限制。

可以通过 GUI 上的 PrintAST 按钮查看生成的 AST 效果。

```
[Program]
    [FunctionDefi]
        [IntType]
        [main]
        [DeclList]
        [CompoundStat]
            [DeclList]
            [StatList]
                [ExpressionStat]
                    [FunctionCall]
                        [Variable]
                        [printf]
                        [ArgumentList]
                            [StringConst] [hello world!]
                [ReturnStat]
                    [IntConst] [0]
```

图 4：GUI 输出的 helloworld 程序的 AST

2.4 特性：复杂类型系统

LoliCCompiler 所支持的类型系统由五个部分组成：

基本类型：int、char、void。

指针类型：修饰某个具体类型。

数组类型：修饰某个具体类型，并具有固定的数组大小。

记录类型：由若干个具体类型按照固定顺序组合。

函数类型：由一个具体类型作为返回值，若干个具体类型按照顺序作为参数。（注：类型表示支持变参函数但仅用于内置 printf/scanf，语法并不能识别变参函数。）

其中除了基本类型以外的四种类型可以进行任意的嵌套组合。

这一特性的实现有两点：一是复杂类型的表示，二是复杂类型的识别。

复杂类型的表示通过面向对象的继承方法是容易设计的。而复杂类型的识别较为复杂，这是因为 C 语言的类型修饰之中指针是前缀修饰，数组和函数后缀修饰，而优先级却是以后缀修饰优先，并且函数定义的括号与改变类型修饰优先级的括号容易引发语法冲突。于是需要精妙设计相应的上下文无关文法加以解决。

```
char *(*(**foo[][8])())[];
```

```
[VariableDecl]
    [ArrayType]
        [ArrayType]
            [PointerType]
                [PointerType]
                    [FunctionType]
                        [ReturnType]
                        [PointerType]
                            [ArrayType]
                                [PointerType]
                                    [CharType]
                        [Paras]
            [IntConst] [8]
    [foo]
```

图 5：C 复杂类型定义和对应的 AST

2.5 特性：用户友好型 GUI

LoliCCompiler 相比其他甚至于企业级编译器而言的一大特色就是有着一个设计简洁且非常用户友好的 GUI。

左上是控制面板。有着包括打开/保存在内的各种功能按钮。可以输出各种中间表示。

左下是编译信息。出现语法或者语义错误时会在这里显示错误提示信息。

右侧则是输出文本框。对应的输出会显示在这大片区域中。

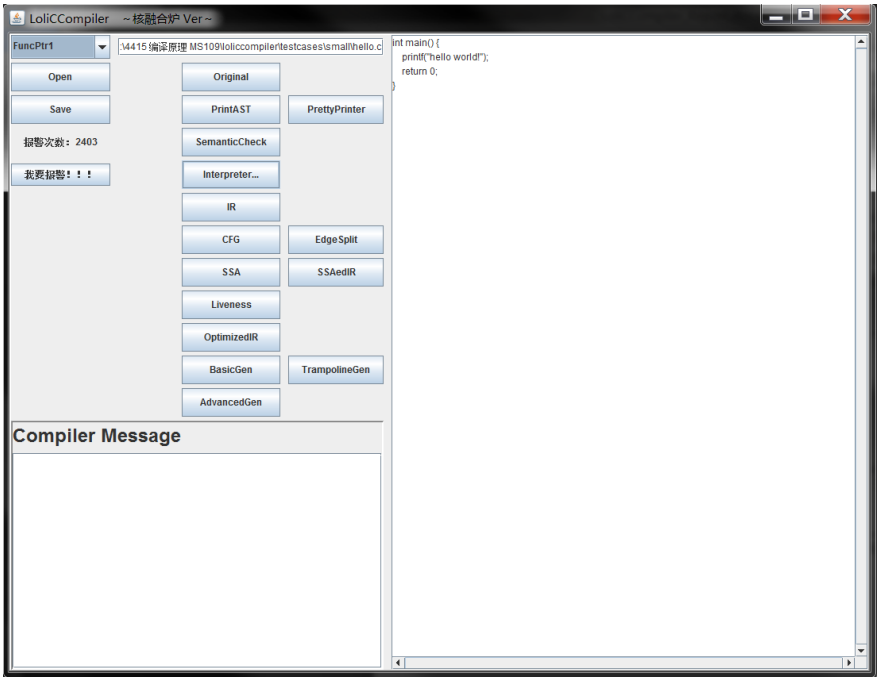

图 6：GUI 主界面

使用 java 的 GUI 库 java.swing，实现并不复杂。通过这一 GUI，可以详细地观察到 LoliCCompiler 执行的中间结果。

2.6 特性：代码美化工具

LoliCCompiler 实现了 Pretty Printer 也就是代码美化工具。其主要功能是将输入的源代码进行合理的缩进并插入/删除空格空行使其变得更为美观，便于阅读。

```
char *_ = "#include <stdio.h>%cchar* recurse=%c%s%c;%cint main(){printf(recurse,10,34,recurse,34,10,10);}%c"
int M[5000]={2},*u=M,N[5000],R=22,a[4],l[5]={0,-1,39-1,-1},m[4]={1,-39,-1,39},*b=N,
*d=N,c,e,f,g,i,j,k,s;int main(){for(M[i=39*R-1]=24;f|d>=b;){c=M[g=i];i=e;for(s=f=0;
s<4;s++)if((k=m[s]+g)>=0&&k<39*R&&l[s]!=k%39&&(!M[k]||!j&&c>=16!=M[k]>=16))a[f++
]=s;if(f){f=M[e=m[s=a[1/(1+2147483647/f)]]+g];if(j < f) j=f;f+=c&-16*!j;M[g]=
c|1<<s;M[*d++=e]=f|1<<(s+2)%4;}else if(d > b++) e=b[-1];}printf(" ");for(s=39;--s;printf("_")
)printf(" "); for(;printf("\n"),R--;printf("|"))for(e=39;e--;printf("%c", *("_ "+(*u++/8)%2)))printf("%c", *("| "+(*u/4)%2
                ));}
```

```
char *_ = "#include <stdio.h>%cchar* recurse=%c%s%c;%cint main(){printf(recurse,10,34,recurse,34,10,10);}%c

int M[5000] = {2}, *u = M, N[5000], R = 22, a[4], l[5] = {0, -1, 39 - 1, -1}, m[4] = {1, -39, -1, 39}, *b = N, *d = N, c, e, f, g

int main() {
  for (M[i = 39 * R - 1] = 24; f | d >= b; ) {
    c = M[g = i];
    i = e;
    for (s = f = 0; s < 4; s++)
      if ((k = m[s] + g) >= 0 && k < 39 * R && l[s] != k % 39 && (!M[k] || !j && c >= 16 != M[k] >= 16))
        a[f++] = s;
    if (f) {
      f = M[e = m[s = a[1 / (1 + 2147483647 / f)]] + g];
      if (j < f)
        j = f;
      f += c & -16 * !j;
      M[g] = c | 1 <<= s;
      M[*d++ = e] = f | 1 <<= (s + 2) % 4;
    } else
      if (d > b++)
        e = b[-1];
  }
  printf(" ");
  for (s = 39; --s; printf("_"))
    printf(" ");
  for (; printf("\n"), R--; printf("|"))
    for (e = 39; e--; printf("%c", *("_ " + *u++ / 8 % 2)))
      printf("%c", *("| " + *u / 4 % 2));
}
```

图 7：代码美化前后效果对比

(使用的数据是 Phase2/Passed-new/madcalc.c)

实现上，使用了一个访问者 prettyprinter 来遍历整棵 AST，一边维护缩进信息一边输出到缓冲区中。而利用缓冲区处理类型的前后缀修饰等需要调整输出顺序等情况。

美中不足的是，由于是在 AST 进行的遍历，丢失了源代码当中的注释和括号等信息。LoliCCopmiler 是采用对于运算符优先级的运算来加上必要的括号来保证语义等价性的。

3 Semantic Check

完成抽象语法树的建立之后，需要对于语法树进行语义检查，这是 LoliCCompiler 的第二个模块。语义检查的目的主要有三点：一是检测出无法被编译的、具有语义错误的代码；二是

使得代码的表示变得更为抽象化，忽略语法结构细节，便于进一步的编译操作；三是对于变量大小等信息进行计算，为进一步的编译提供必要信息。我选择了通过将抽象语法树转化为中间表示树的方法来实现这三个目的。

3.1 中间表示树设计

中间表示树(IRT)是 LoliCCompiler 对于源程序的第二个等价描述。其主要目的是更加抽象地描述程序并聚合后面编译步骤所需要的信息。

我所设计的 IRT 仅有三个模块：声明、语句、表达式。其中声明只有一个类，即是变量声明，函数和类型的声明经过语义检查都已经不再需要，语句还是保留了程序的控制流，有循环、分支、跳转、表达式以及组合语句五个子类，而表达式也只有一个类，通过工厂模式对于运算符类进行封装，这使得在之后的中间代码生成之中，可以由各个运算符自己处理自己的翻译方法。

至于 AST 中的类型和初始化列表，经过语义检查，其中必要的信息也已经保留到了 IRT 各个类的属性之中。

IRT 的类图如下。Program 是 IRT 的根。可以看出相对 AST 已经有了很大的简化。

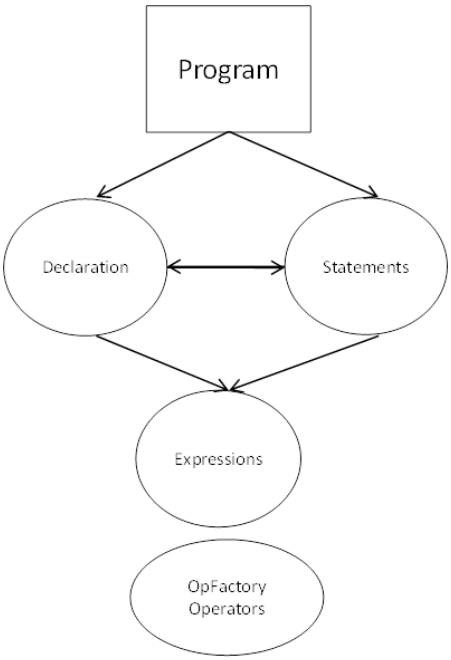

图 8：IRT 类结构图

3.2 语义检查实现

语义检查仍然是通过一个 visitor 来遍历 AST，并且在这个 visitor 当中维护变量、函数、类型等的环境信息来实现的。在语义检查的过程中利用一个栈来建立 IRT。

语义检查有很多的实现细节，这是为了支持繁多的语言特性。我觉得这并没有特别好的实现方法，还是要依靠对于各种语义的枚举亦或是对于不同情况的分类讨论。例如减法对于两个操作数分别是不是指针就有完全不同的操作，必须进行分别处理。而+=这类运算赋值运算符也不能简单地拆成+和=先运算再赋值。

我通过工厂模式对于运算符以及运算符的构造过程进行了封装，将所有的讨论都限定在了

语义检查的阶段，之后的阶段就不需要再枚举是什么运算符再来进行翻译了，这是因为这一数据信息已经被转化为了更加便于使用的结构信息，包含在了对象本身之中。

```
232     private OpFactory calPost(int op) {
233         switch (op) {
234             case Symbols.INC_OP : return Factories.POINC.getFact();
235             case Symbols.DEC_OP : return Factories.PODEC.getFact();
236             default : throw new InternalError("Unexpected Post operator.\n");
237         }
238     }
239
240     void define(int id, Type type) { define(id, type, VariableTable.VARIABLE); }
243
244     void define(int id, Type type, int isVari) {
245         if (table.checkCurId(id)) {
246             if (!isGlobal() || !typeEqual(type, table.getId(id)) || table.checkType(id) != isVari) {
247                 throw new SemanticError("Identifier " + id + " redeclared as a different kind of symbol.\n");
248             } else if (table.checkDefi(id)) {
249                 throw new SemanticError("Identifier " + id + " redefined.\n");
250             }
251         } else {
252             if (isVari == VariableTable.VARIABLE) {
253                 table.addVari(id, type);
254             } else {
255                 table.addType(id, type);
256             }
257         }
258         table.defiVari(id);
259     }
```

图 9：IRTBuilder.java 代码片段

3.3 特性：语法/语义错误信息

LoliCCompiler 有着完整的异常处理机制。其异常类 CompileError 继承 RuntimeException。有四个子类：

InternalError：编译器内部错误，属于 bug。

InterpretError：解释器运行时错误，包括除 0 等。

SemanticError：语义检查发现的语义错误。

SyntacticError：Lexer 或 Parser 发现的语法错误。

对于在程序中发现的语义或语法错误，LoliCCompilerGUI 可以捕捉到被抛出的异常，并显示在 CompilerMessage 区中。有数十种语法和语义错误提示，仿照了 GCC 的风格。

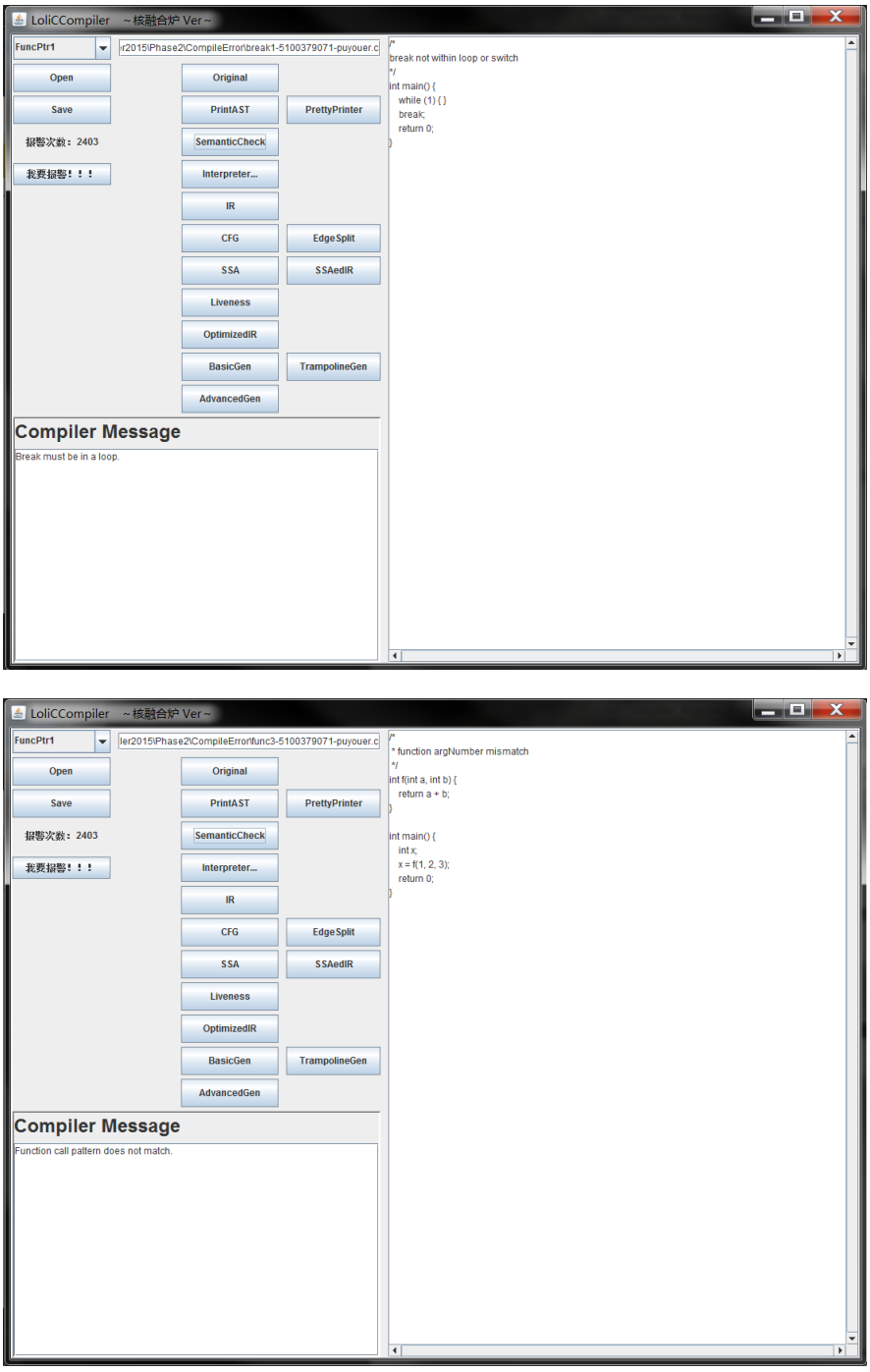

图 10、11 编译错误信息实例

3.4 特性：未定大小数组

在 C 语言中，未定大小数组共有三种：1、可以由初始化列表计算出数组大小。2、将数组作为一个函数的参数进行传递。3、对于 struct 的最后一个元素，如果是数组，可以是可变大小数组，其实际大小将由被分配到的内存大小决定。

LoliCCompiler 支持所有以上三种未定大小数组的使用方式。主要分为三个步骤：1、在语法分析阶段支持大小为空的数组。2、对于可由初始化列表计算得到的，在语义检查阶段进行计算。对于参数，将其看作一个相应类型的指针。3、在解释器中正确模拟以及在中间代码和

代码生成中正确实现对应的内存分配和地址访问。

3.5 特性：typedef 支持

typedef 是 C 语言所提供的一个语法的糖衣。通过 typedef 可以为复杂类型提供较为方便使用的别名。一个常见的做法就是将一个记录名 typedef 为对应的记录类型。

虽然在实际中很少会有人将 typedef 过的类型名再重新使用为变量名，但这对于编译器而言是必须要考虑的 corner case。

LoliCCompiler 对于 typedef 的实现主要是基于环境的 lexer hacking 和语义检查阶段的支持。

3.5.1 Lexer Hacking

分词器(Lexer)无法判断一个 Identifier 是否是被 typedef 过的 Typename。然而语法分析器(Paser)却依赖分词器所给出的符号来进行语法分析。于是，必须有某种方法让语法分析器“告诉”分词器现在某个特定的 Identifier 是否是被 typedef 过的类型名。这种反信息流而行的方法被称为 Lexer hacking。

LoliCCompiler 也实现了 Lexer hacking 的技术。在分词器和语法分析器之间搭建了一个环境，即一张符号表，分词器根据这一符号表的信息判断某一个 Identifier 是类型名还是标识符，而语法分析器会根据语法规则的语义添加删除名字空间，并将某一标识符判定为类型名加入到符号表中。小心地实现这一过程，可以解决类型名和标识符的混淆问题。

3.5.2 语义检查支持

经过语法分析，typedef 会被看成声明被放入 AST 中，而在语义检查阶段，建立了基于语义的符号表来对 DefinedType 进行解定义，还原回其原类型。因此 typedef 对解释器及后面的编译过程就不再产生影响了。

3.6 特性：C 语言解释器

LoliCCompiler 的一项特色功能是其内置的 C 语言的解释器 LoliCInterpreter。可以直接运行 C 语言源程序。其将源程序整个读入，在进行完语义检查之后得到的 IRT 上对其运行过程进行模拟，通过模拟控制台的 GUI 与用户进行交互。

LoliCInterpreter 本身是 IRT 的一个 visitor，由于对于表达式的计算封装在了运算符类之中，解释器主要实现控制流、堆栈存储和 STL。

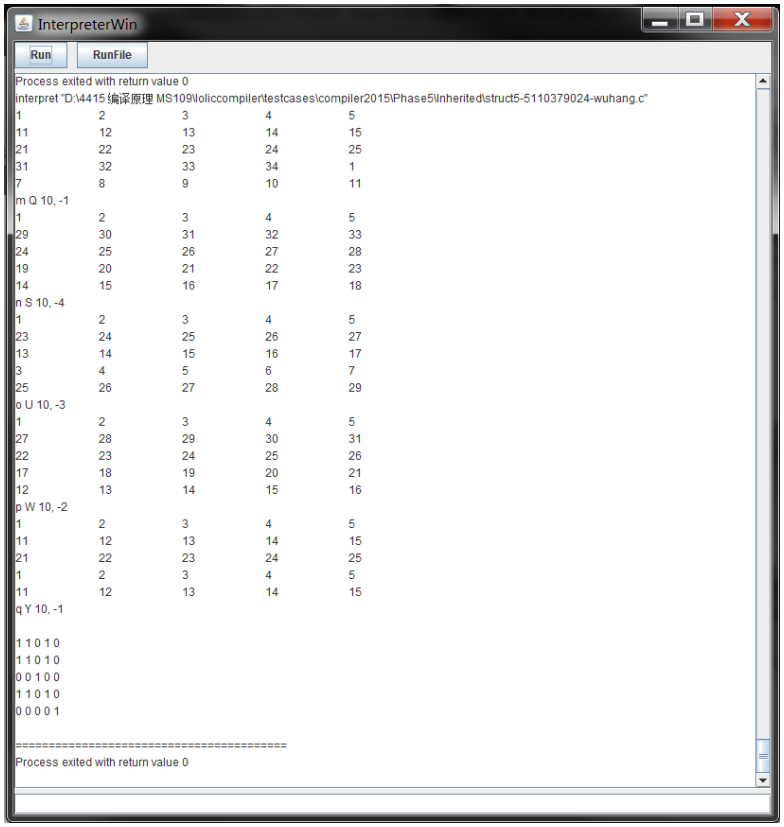

图 12：解释器 GUI

(图中显示的是 struct5.c 的运行结果，正确无误)

3.6.1 控制流实现

LoliCInterpreter 本身是 IRT 的 visitor，对于某一 IRT 的节点，让 Interpreter 对其进行访问即是进行对这一节点的解释。因此，函数调用的控制流恰好由系统的调用来实现，而分支和循环的实现也相对简单，对于 break 和 continue 则在 Interpreter 设置了一个中断信号，有中断信号时，将不会对语句进行解释，只有特定能够改变信号并对其作出反应。

3.6.2 模拟堆栈

LoliCInterpreter 使用了一个大小为 4MB 的 byte 类型的数组来模拟实际运行时的堆栈。处理上有三个要点：

1、正确处理函数调用和返回。模拟了运行时环境的栈帧，在调用和返回时会进行帧的分配和删除。

2、栈空间和堆空间。栈和堆分别从 byte 数组的两端开始。malloc 将从末端开始分配空间。而栈则由栈顶指针从头端开始进行空间的分配。

3、地址和指针。取地址操作将会得到虚拟堆栈上的地址，而指针运算则按照实际在内存中那样进行，解释器能够正确实现包括函数指针在内的指针访问。

3.6.3 语言特性支持：未定大小数组/函数指针/STL(含 scanf)支持

typedef 的问题在语义检查阶段就已经解决了。而未定大小数组则仅剩下实际是对分配空间和寻址访问的支持，在模拟堆栈的基础上也是较为容易的。

对于函数指针，LoliCInterpreter 无法像实际的函数指针那样保存一个指令在内存中的地址，而是保存的所调用的函数的 id，当通过函数指针引发调用时，将在环境中寻找对应 id 的函数来进行访问。

对于 STL 的支持，LoliCInterpreter 用 Java 严格按照 C 的标准进实现了 printf/ getchar/ malloc/ scanf。其输入和输出都建立了一层缓冲区。malloc 移动并返回堆顶指针。getchar 取出缓冲区的下一个字符。printf 则根据格式描述符来依次输出每个参数。scanf 则从输入读入字符串并按照格式描述符进行解释放到作为参数的每个地址中。

而为了模拟实时响应的控制台，采用了 Java 多线程技术，在不同的线程中运行解释器和处理 IO 中断。解释器同样支持一些较为简单的运行时错误，会显示在模拟控制台中。

4 Intermediate Representation

将中间表示树转化为中间表示，即三地址代码，是 LoliCCompiler 的第三个模块。中间表示已经完全消除了源代码的语法特性的细节，复杂操作也被分解为可以被翻译为汇编代码的多个简单操作，开始接近可以实际运行的机器代码。而大部分的优化实质上则是从一种中间代码到同一种或另一种中间代码的映射，所以这一步骤对于生成高质量的机器代码而言是必要的。我在这一阶段除了使用三地址代码以外，还实现了静态单赋值(SSA)作为一种辅助中间表示。

4.1 中间代码设计和转化

中间代码开始考虑实际对汇编代码的翻译，因而相对于 IRT 来说要更加具体一些。LoliCCompiler 所使用的中间代码采用的是三地址代码。抽象父类 MIRInst，有这样一些子类：

AssignInst：a = b op c 赋值指令。

CallInst：a = func z 以前 z 条 ParaInst 作为参数调用函数 func(可以是函数指针)并将返回值附给 a 的调用指令。

GotoInst：Goto label 无条件跳转指令。

IfInst：if a relop b then label 分支指令。

MemInst：a ←mem[b] 局部和全局变量内存空间分配指令。

ParaInst：para a 保留 a 将作为参数使用

RecvInst：recv a 声明 a 作为参数被传入

ReturnInst：return 返回调用者

此外有两类特殊的指令：

PhiInst：ai=phi(ai1,ai2,…) SSA Phi 函数

TrampInst：tramped_func = trampoline func 为了实现高阶函数而插入的 Trampoline 指令。

而在对于变量和值的处理上采用了这样的面向对象方法：

Value：抽象类任意值，以下所有类均继承 Value。

Const：抽象类表示常量，子类包括整数、字符和字符串。

VarName：表示变量。可以是临时变量或者用户定义的变量。
DeRefVar：继承 VarName。指针解引用得到的变量。
SSAVarName：继承 VarName。SSA 中使用的具有 SSA 标号的变量。

而从 IRT 到中间表示的转化也并不十分复杂。依据每个 IRT 节点的类型和信息作相应的翻译即可。难点在于对于标号的处理。为了避免生成冗余标号，LoliCCompiler 的中间表示生成结合了龙书上的标号回填和伪标号方法。一个标号除非被某个跳转指令使用，否则就会被认为是伪标号，在中间代码结束后就会被删除。而对于那些生成时标号还不能确定的跳转指令 break/continue，则先将其生成，并放入等待列表之中，由对于循环翻译来回填标号。

```
343     public List<MIRInst> gen(Label cur, SeSt st, Label next) {
344         List<MIRInst> list = new LinkedList<>();
345         if (st.fl instanceof ExSt && ((ExSt)st.fl).expr.op instanceof IntOp) {
346             Label iftr = new Label(Label.FALL | Label.DUMMY);
347             list.addAll(genRel(cur, st.expr, iftr, next));
348             list.addAll(genStat(iftr, st.tr, next));
349         } else {
350             Label trjp = new Label(Label.DUMMY), iftr = new Label(Label.FALL | Label.DUMMY), iffl = ne
351             list.addAll(genRel(cur, st.expr, iftr, iffl));
352             list.addAll(genStat(iftr, st.tr, trjp));
353             boolean flag = false;
354             if (next.isFall()) {
355                 flag = true;
356                 next.st |= Label.FALL;
357             }
358             list.add((new GotoInst(next)).setLabel(trjp));
359             if (flag) {
360                 next.st |= Label.FALL;
361             }
362             list.addAll(genStat(iffl, st.fl, next));
363         }
364         return list;
365     }
```

图 13：MIRGen.java 代码片段

4.2 特性：控制流优化

LoliCCompiler 在中间表示生成阶段针对 if 分支语句的翻译进行了特别的优化，能够大量减少冗余的跳转指令。这一优化是龙书上一道思考题。其主要原理在于利用 if 条件为假时将继续执行下一行语句的性质，在标号上打上 fall 标记，即表明如出现这一分支应继续执行下一条语句而无须跳转，并且当 fall 标记出现在条件满足时，利用关系表达式的恒等变换(取否)，并交换 true 和 false 时的标号来最大程度地利用这一点。这使得 LoliCCompiler 对于具有复杂条件的判断能生成很优秀的代码。例如数据中的 superloop.c 在最里层循环中的 if，LoliCCompiler 当其中任一条件不满足时立即跳转至循环末尾，效率极高，即使使用最暴力的代码生成方式也能轻松通过优化门限。

```
for ( a=1; a<=N; a++ )
for ( b=1; b<=N; b++ )
for ( c=1; c<=N; c++ )
for ( d=1; d<=N; d++ )
for ( e=1; e<=N; e++ )
for ( f=1; f<=N; f++ )
        if (a!=b && a!=c && a!=d && a!=e && a!=f && a!=h && a!=i && a!=j && a!=k
&& b!=c && b!=d && b!=e && b!=f && b!=h && b!=i && b!=j && b!=k
&& c!=d && c!=e && c!=f && c!=h && c!=i && c!=j && c!=k
&& d!=e && d!=f && d!=h && d!=i && d!=j && d!=k
&& e!=f && e!=h && e!=i && e!=j && e!=k
&& f!=h && f!=i && f!=j && f!=k && i!=j && h!=k)
        {
                total++;
        }

printf("%d\n", total);
```

```
if a_1 == b_1 then _L14
if a_1 == c_1 then _L14
if a_1 == d_1 then _L14
if a_1 == e_1 then _L14
if a_1 == f_1 then _L14
if a_1 == h_0 then _L14
if a_1 == i_0 then _L14
if a_1 == j_0 then _L14
if a_1 == k_0 then _L14
if b_1 == c_1 then _L14
if b_1 == d_1 then _L14
if b_1 == e_1 then _L14
if b_1 == f_1 then _L14
if b_1 == h_0 then _L14
if b_1 == i_0 then _L14
if b_1 == j_0 then _L14
if b_1 == k_0 then _L14
if c_1 == d_1 then _L14
if c_1 == e_1 then _L14
```

图 14、15：superloop.c 源代码和翻译得到的中间代码（部分）

4.3 特性：函数指针

函数指针是 C 语言的一个重要语言特性。使用函数指针可以将函数作为参数传给其他函数，用于实现一些简单的多态，例如排序算法使用不同的比较和交换函数，同时可以提供接口而一定程度上封装实现细节。

LoliCComplier 和 LoliCInterpreter 支持函数指针的赋值和调用，但对于函数指针强制转换后解引用进行赋值等则被视为未定义行为，将会引发无法预见的结果(SPIM 会抛出异常静态代码区内存无法写入)。

具体实现上，LoliCCompiler 在语法阶段支持了复杂类型，在语义检查阶段支持了函数指针的赋值以及调用，在中间表示和代码生成中仅仅是将调用标号改为调用变量。然而引入函数指针会对过程间控制流和数据流分析带来极大的困难。

4.4 特性：内嵌函数和高阶函数

内嵌函数是 C 语言的一个语言特性，但在 C++里被废弃了。其内容是支持在一个函数体内定义另一个函数，且这个被定义的函数只能在其名字空间内被访问，或者通过函数指针传递出去被访问。问题在于内嵌函数可以使用于其定义时可见的所有变量，而当一个内嵌函数被函数指针传出时必须仍然保持这一特性。这就涉及到了这些变量的生存周期问题。

有三种处理方法：1、不支持使用这些变量(C99)。2、允许在这些变量原本在栈上的生存周期内使用(gcc)。3、允许在这个内嵌函数的函数指针存在的生存周期内使用(clang)。其中，第三种做法需要实现函数闭包，对于堆栈结构要做出较大改动，工程量过大，未能实现。而所谓高阶函数，是指像处理数据那样处理函数的语法特性，按照这一定义，实际上函数指针也能算作是高阶函数。但为了体现出高阶函数这一特性和函数指针特性的区别，LoliCCompiler 实现了内嵌函数，并采用了与 gcc 相同的第二种做法，一种被称为 Trampoline 的处理技术。

Trampoline 技术的关键是在外层函数的栈上写入一小段被称为 Trampoline 的机器代码，用于为其内嵌函数设置好用于访问其局部变量的静态栈指针。在中间表示阶段，LoliCCompiler 使用了特殊的 TrampInst 语句来体现这一特性。

在第四模块代码生成中更加详细地探讨了具体实现细节以及所遇到的在 SPIM 不支持动态代码的情况。

4.5 特性：静态单赋值(SSA)

在 LoliCCompiler 中，还采用了被称为静态单赋值(SSA)的中间表示形式用于优化。SSA 是较为新颖的一种中间表示。严格地来说，SSA 并不是某种特定的中间表示形式，更接近于一种性质，其主要的特点是对于变量进行重命名，使得每个变量只会被赋值一次，然后被多次使用不再被修改。从数据流分析的角度来看，实际上是建立了隐式的定义-使用链，即把变量的某次定义与这次定义的使用通过重命名体现了出来。

而 SSA 有什么样的作用呢？主要是对于一些优化有所帮助。例如常数传播、公共子表达式消除、冗余复制消除、（激进）死代码消除、稀有条件传播、值标号等数据流优化，良好实现的 SSA 中间表示能够使得这些优化的实现变得更为简单，运行更加高效，优化的结果也更好，这是因为 SSA 较为明确地计算出了定义-使用链。对于较大规模的编译来说则更能体现出 SSA 的优势，因而 SSA 被很多现代编译器如 gcc 等作为中间表示的一部分。LoliCCompiler 中基于 SSA 的优化实现将会在第六节优化中专门讨论。

将中间代码转化为 SSA 的过程并不十分简单。其所要处理的主要问题是 phi 函数的处理。考虑在一个 if 的两个分支之中给一个变量赋不同的值，按照 SSA 的性质，那在这两个分支汇合之后的变量应该是谁呢？这时就需要 phi 函数来将两个变量进行合并。请注意，SSA 所使用的 phi 函数并不是真正存在的、能够被翻译为相应的机器代码的（如果强行要翻译，则应该是判断是从哪个入口进入的，再进行值的拷贝，这样的大量操作使优化失去了意义），我们仅是通过 phi 函数这一记号来表明这些变量之间的关系，并会在做完优化之后消除掉这些 phi 函数再作进一步的编译。

接下来分为 8 个部分来介绍 LoliCCompiler 中 SSA 的实现。

4.5.1 控制流分析、控制流图、边切分性质

对于三地址代码中间表示的每一个过程进行控制流分析，建立控制流图是 SSA 转化的准备工作。LoliCCompiler 采用了传统的标明块首-块切分-连边的控制流分析方法构建控制流图。

为了能够方便而高效地地将 SSA 转化回中间表示，我们需要对控制流图进行处理，使之具有边切分性质。即每个控制流图中的顶点要么有单一前驱，要么有单一后继。算法是对于每条连接从一个有多后继到多前驱的边(u,v)，建立一个新定点 z，将其切分为(u,z)和(z,v)两条边。这需要在 u 最后跳转至 z，并在 z 中跳转至 v。如果最后生成了多余的跳转，我们也可很方便地将其优化掉。

4.5.2 最近支配点计算

转化 SSA 的第二步准备工作是计算最近支配点，又称直接支配点。所谓一个点 u 支配另一个点 v，即是到 v 的每一条路径都必须经过 u，而 v 最近支配点是所有支配 v 的点中在控制流图的某个 dfs 树上离 v 最近的祖先。这样一种最近支配点的关系会形成一棵树结构，被称为支配者树，而这正是 SSA 的计算所需要的。

计算最近支配点有暴力迭代算法和使用并查集的 Lengauer-Tarjan 算法，LoliCCompiler 实现了后者。

4.5.3 支配边界计算

在得到每个点的最近支配点，也就是建立出支配者树之后，需要进行 SSA 的第三部准备工作，即是支配边界(Dominance Frontier)的计算。所谓一个顶点 u 的支配边界，是这样的点 v 的集合，存在从 u 开始的一条路径到达 v，但 u 并不支配 v，即存在另一条路径可以不经过 u 到达 v，并且 u 还支配了 v 的某个前驱（这体现了“边界”的含义）。这样我们就需要在 v 的开头插入 phi 函数合并 u 中的 SSA 变量和其他路径到达 v 的 SSA 变量。

支配边界的计算按照逆支配者树的顺序进行一遍迭代即可。

4.5.4 插入 phi 函数

计算完支配边界，就可以开始真正对于 SSA 的转化了。按照前面支配边界的分析，u 如果能够到达且支配了 v 的某些个前驱却又不支配 v，就需要在 v 插入 u 中变量的 phi 函数。按照这一规则在 v 中进行插入在 u 中定义的变量即可。

4.5.5 变量重命名

在必要的位置插入 phi 函数之后，就需要进行 SSA 变量的重命名。从过程入口开始 dfs 遍历支配者树，遇到定义就新申请一个 SSA 变量名，遇到使用就将变量名进行替换，对于其控制流图上的后继如果存在这一变量的 phi 函数，也用现在最新的变量名进行替换，在基本块结束时把所有新加入的 SSA 变量一并弹出。这样即可完成 SSA 变量的重命名，SSA 的转化就算基本完成了。

4.5.6 活性分析修剪

然而按照上面的方式插入 phi 函数实际上会插入很多不必要的 phi 函数。考虑一个变量仅在一个基本块内被定义使用，但却会在所有这个基本块的支配边界中插入关于这个变量的 phi 函数，这些冗余的 phi 函数是我们所不需要的，因此需要进行 phi 函数的修剪。修剪可以在插入时修剪也可以在插入后修剪。

能够删除全部冗余 phi 函数，即全修剪的一种方法是对于每个过程进行活性分析。当某个变量不在某个支配边界的 livein 集合中，便没有必要再这个支配边界插入关于这个变量的 phi 函数。

LoliCCompiler 实现了基于活性分析的全修剪。

4.5.7 转换回中间代码

在 SSA 上做完优化之后，我们需要将其转化回三地址代码来进行进一步的编译。这时需要消除 phi 函数，并把变量名改回成非 SSA 的变量名。一种简单粗暴的做法是直接将每个变量改回其原本的名字，但这样会丢失 SSA 的良好性质。较为高效的做法是利用之前提到的边切分性质，对于每个块开头的 phi 函数，在其相应的前驱中插入一条相应拷贝语句，并删除这一 phi 函数。边切分性质的好处在于它可以减少冗余的赋值，例如一个后继使用 a1 而另一个后继并不使用，就没有必要再所有的后继之前先复制 a1 的值。

LoliCCompiler 就是在基于边切分的控制流图上利用上述做法将 SSA 转换回三地址代码的。

4.5.8 别名处理

在 SSA 中，还有一个问题，即所谓“别名”。例如 C 语言通过指针来访问变量，就无法确定指针指向的究竟是哪一个变量。如果不考虑指针访问的情形，直接做 SSA 就会发生问题。比较简单的处理方法是直接不对那些可能存在别名的变量做 SSA。这也是 LoliCCompiler 所采用的方法。而更为高效的做法则是基于别名分析再进行 SSA，尽可能地保留可以被做 SSA 的部分。

5 Code Generation

LoliCCompiler 的最后一个模块是将中间表示转化为汇编代码。这一阶段所要解决的问题是代码选择、寄存器分配和指派。我除了实现了最为基本的不使用寄存器而依赖内存的暴力代码生成以外，还实现了较为复杂但效果卓越的跨过程全局寄存器指派和分配。

5.1 暴力代码生成

最简单的 MIPS 代码生成方法，就是对于每条中间表示指令先将所有需要的信息读到寄存器当中，然后计算，再将结果存回内存。在函数调用的时候也直接将参数按照顺序写在栈上然后移动栈指针即可。这样的代码生成方法固然效率不高。LoliCCompiler 实现了这一方法主要用于检验正确，并基于这一方法支持函数指针及高阶函数。

```
353     void genInst(CallInst inst) {
354         alignTo(4);
355         curDelta += 4;
356         code.addText("\t\tsw\t$ra\t" + -curDelta + "($fp)");
357         int bak = curDelta;
358         LinkedList<Value> tmp = new LinkedList<>();
359         for (int i = 0; i < ((IntConst)inst.num).val; ++i) {
360             tmp.push(pStack.pop());
361         }
362         for (Value val : tmp) {
363             alignTo(val.align);
364             if (val instanceof VarName && ((VarName) val).isArray) {
365                 curDelta += 4;
366                 code.addText("\t\t" + "sw\t" + loadToReg(val).toString() + "\t" + -curDelta + "($fp)");
367             } else {
368                 curDelta += val.size;
369                 if (val instanceof VarName && ((VarName) val).isStruct) {
370                     if (val.align == 1) {
371                         BasicReg addr = loadToReg(val), tmpr = getReg();
372                         for (int i = 0; i < val.size; ++i) {
373                             code.addText("\t\t" + "lb\t" + tmpr.toString() + "\t" + i + "(" + addr.toStri
374                             code.addText("\t\t" + "sb\t" + tmpr.toString() + "\t" + (-curDelta + i) + "(
375                         }
376                     } else {
377                         BasicReg addr = loadToReg(val), tmpr = getReg();
378                         for (int i = 0; i < val.size; i += 4) {
```

图 16：BasicGen.java 代码片段

5.2 高效代码生成

更加高效的代码生成则需要更加充分地利用目标机的资源，包括寄存器和一些特殊指令例如加一个立即数和较为复杂的寻址模式等。除了使用全局寄存器分配，LoliCCompiler 的高效代码生成对于立即数和寻址模式做了较为细致的优化，并且实现了寄存器合并、活性分析剪枝、STL 内联展开三项低级抽象表示的优化，而对于使用寄存器可能出现的指针访问问题也进行了小心的处理。

```
1044                Value val = inst.src2;
1045                if (val instanceof IntConst) {
1046                    LinkedList<SPIMRegister> list = new LinkedList<>();
1047                    list.push(loadToReg(inst.src1));
1048                    list.push(writeResultReg(inst.dest));
1049                    code.addText(new SPIMInst(SPIMOp.valueOf(inst.op.name()), list.get(0), list.get(1),
1050                } else if (val instanceof CharConst) {
1051                    LinkedList<SPIMRegister> list = new LinkedList<>();
1052                    list.push(loadToReg(inst.src1));
1053                    list.push(writeResultReg(inst.dest));
1054                    code.addText(new SPIMInst(SPIMOp.valueOf(inst.op.name()), list.get(0), list.get(1),
1055                } else if (val instanceof StringConst) {
1056                    LinkedList<SPIMRegister> list = new LinkedList<>();
1057                    list.push(loadToReg(inst.src1));
1058                    list.push(writeResultReg(inst.dest));
1059                    code.addText(new SPIMInst(SPIMOp.valueOf(inst.op.name()), list.get(0), list.get(1),
1060                } else if (val instanceof VarName && (((VarName) val).isStruct || ((VarName) val).isA
1061                    if (val instanceof DeRefVar) {
1062                        val = ((DeRefVar) val).val;
1063                    }
1064                    LinkedList<SPIMRegister> list = new LinkedList<>();
1065                    list.push(loadToReg(inst.src1));
```

图 17：AdvancedGen.java 代码片段

5.2.1 寄存器合并

寄存器合并是出现形如 x=y 的复制语句时，并不生成将 y 写入到 x 的地址或是拷贝至 x 的寄存器的指令，而是在 y 的寄存器描述符上加上 x 的标记。当某个寄存器需要被写入或其中内容需要更新时，也不一定将其描述符中所有变量全部写回内存而是当描述符中变量较多时拷贝至一新寄存器内。这一优化可以省去不少的存取指令。

5.2.2 活性分析剪枝

通过对于中间代码进行精确到每条语句的活性分析，寄存器描述内的变量如果没有出现在 liveout 集合当中，即不再会被使用便不会再被存回内存或放入其他寄存器中，这一优化对于局部临时变量非常有效。

5.2.3 STL 内联展开

对于 getchar/malloc 以及 printf 的部分特例，LoliCCompiler 进行了 STL 函数的内联展开，不再按照函数调用对其进行处理而直接在原地进行 syscall。减少了一定的指令数。

5.2.4 别名处理

由于指针访问直接访问内存，将变量完全保存在寄存器中可能会引发问题，应对这一情况，LoliCCompiler 对于指针进行了较为粗略的分析，将可能被指针访问的变量及时在内存和寄存器之间进行了同步。

5.3 特性：全局寄存器指派和分配

LoliCCompiler 高效代码生成的核心是基于循环分析和过程间分析的全局寄存器分配。主要分为四个步骤：

1、对中间表示进行循环分析评估变量的权重以及过程间分析明确过程之间相互调用的关系。通过这两部分的信息确定某些变量为绑定寄存器变量，在基本块之间将保持在同一个寄存器中。

2、使用上文提到的方法进行符号寄存器代码选择。得到了一个非常接近实际 SPIM 汇编代码的低级中间表示。

3、基于过程分析和在低级中间表示上的活性分析的结果构建符号寄存器之间的冲突图。并对冲突图进行图染色算法。得到符号寄存器到物理寄存器一个映射。

4、根据这一映射替换原来低级中间表示中的符号寄存器并在必要的位置插入处理溢出的代码。生成最终汇编代码。

这一套方法可以高效利用寄存器，生成较为优秀的汇编代码。下面就具体实现细节做简单介绍。

5.3.1 循环分析

通过控制流图以及支配者树可以识别中间表示中的自然循环。方法是从 DFS 树中的回边开始进行标记。绝大部分的程序的循环要么一个被另一个包含，要么没有交集，这对于我们分析一个变量的使用情况提供了非常好的性质。

LoliCCompiler 根据一个变量所处的循环的层数、定义和使用情况来对每个变量进行估价，以确定是否要将其指定为绑定寄存器变量。

5.3.2 过程间分析

通过过程间相互调用的关系建立控制流图，形成环的过程将会被合并为一个顶点，因此整个图形成了有向无环图的结构。在寄存器分配时，基于过程间分析的结果，如果被调用者的某一变量的寄存器不会被被调用者及其所调用的过程所使用，便不用再调用前后进行存取操作，由此可以大量减少由函数调用造成的不必要的内存访问。

然而函数指针将会使得过程之间相互调用的关系变得难以分析。而 LoliCCompiler 的高效代码生成目前还不能很好地分析对于使用函数指针的调用，暂时采用了保存所有活性变量的做法。

5.3.3 符号寄存器代码选择

主要使用了 5.2 节所提到的方法进行代码选择。与一般的代码选择所不同的是，生成的代码中可以使用假设有无限个的符号寄存器，这个步骤实际上是寄存器的指派，脱离了变量，将分析的对象转化为了符号寄存器。将由后面的步骤进行寄存器的分配，即将符号寄存器转化为物理寄存器的步骤。

5.3.4 图染色寄存器分配

图染色的寄存器分配算法分为三个步骤：1、基于过程间分析和对于符号寄存器代码进行活性分析的结果，判断哪些符号化寄存器不能被分配同一个寄存器，构建冲突图。2、设有 R 个可用于分配的寄存器，试图对冲突图进行 R 染色。3、对于不能进行 R 染色的情况，选择溢出的寄存器，并插入溢出代码。

一个简单的 R 染色贪心策略是每次找到一个度数小于 R 的点就将其擦除压入栈中。但这会遇到所有点度数均大于等于 R 的情况。LoliCCompiler 采取的做法并不是直接溢出寄存器而是仍然将某一寄存器压入栈中。之后在对栈中的寄存器进行染色时如果发现有顶点无法被染色再考虑溢出。

5.3.5 改进溢出处理

对于溢出的符号寄存器，简单的做法是每次读均从内存读入，每次写均写入内存。而 LoliCCompiler 则使用了 t0 和 t1 作为溢出寄存器。溢出的值将被临时保留在 t0/t1 中，如果需要读取则可直接使用，并且保留运算结果，仅当在有其他变量溢出或者必须写入内存等情况时再进行写入。

5.4 特性：scanf 支持

LoliCCompiler 的暴力代码生成附带了一个 MIPS 汇编实现的 scanf，能够通过 syscall 实现最基本的 scanf 输入功能。

5.5 特性：高阶函数（续）

6 Optimizations

实现一些较为简单的优化算法是 LoliCCompiler 项目的一大目标。除了前面提到的全局寄存器分配外。LoliCCompiler 还实现了以下的优化。

6.1 特性：公共子表达式消除

6.2 特性：冗余复制消除(基于 SSA)

6.3 特性：死代码消除(基于 SSA)

7 Conclusion

LoliCCompiler 虽然只是在两个月的时间内由我一个人仓促完成的一个学生项目，和 gcc 等功能强大的现代编译器毫无可比性，但它仍然实现了完成一个 C 编译器的项目目标，从中可以看出很多现代编译器特性的影子。并且它在实现基本的编译功能之外大胆探索创新，支持了许多超过课程要求的独特特性，例如用户友好型 GUI、C 语言解释器 LoliCInterpreter 等等。对于我这么一个对于计算机并无太多了解的大二学生来说，LoliCCompiler 不失为一件值得称赞的十分优秀的作品。

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman Compilers: Principles, Techniques and Tools (Second Edition). Pearson Education, 2006.

[2] Appel, Andrew W. and Ginsburg, Maia. Modern Compiler Implementation in Java (Second Edition). Cambridge University Press, 1997.

[3] Steven S. Muchnick. Advanced Compiler Design and Implementation. Elsevier Science, 1997.

[4] ISO/IEC 9899:1999 - Programming languages - C (C99)

[5] Gerwin Klein, Steve Rowe, and Regis Decamps. JFlex User's Manual. 2015.

[6] Scott Hudson. JCup User's Manual. 2014.

[7] James R. Larus. Appendix A: Assemblers, Linkers, and the SPIM Simulator.

[8] Kathy Sierra, Bert Bates. Head First Java (Second Edition). O'Reilly Media, 2005.

[9] Lots of Authors. Static Single Assignment Book. Not published yet.

[10] GCC, the GNU Compiler Collection, online documentation.

[11] Ted Yin. CIBIC: C Implemented Bare and Ingenuous Compiler. 2014.

Remarks

最大工程 通宵 10 次左右 时间估计 350h 代码量估计 2.5 万行 收获 3 编译器 java/工程 个人管理 还有很多不足 语言特性匮乏 缺少汇编器/连接器 跨平台 今后的维护 full of buges 更多优化 跨平台集成 更多的学习

感谢课程助教 学长 同学