2022Spring北京大学编译实践lab报告

江枫渔火

关注他

30 人赞同了该文章

序言

2022年春季学期的编译原理修改了Lab的内容，改为由助教自行新开发的项目。Lab的最终目的是实现一个可以将 SysY 语言(C 语言的子集)编译到 RISC-V 汇编的编译器。具体的实现可以分成两个阶段，首先是将 SysY 语言翻译到 Koopa 的中间表示代码，然后从中间代码构建最终的 RISC-V 汇编。

环境配置

首先需要配置相关的环境。

Lab 的运行环境以及需要的工具链已经打包成了镜像，可以运行在 `Docker` 中。拉取镜像使用以下命令：

```
docker pull maxxing/compiler-dev
```

然后就遇到了第一个问题——下载速度非常慢。可以尝试修改源，在 `Docker` 的 设置 —— `Docker Engine` 中将对话框代码替换为如下代码：

```
{
  "builder": {
    "gc": {
      "defaultKeepStorage": "20GB",
      "enabled": true
    }
  },
  "experimental": false,
  "features": {
```

```
    "https://mirror.baidubce.com"
  ]
}
```

然后就可以欢乐下载体验镜像文件了。但是重启之后会遇到 `Docker` 崩溃的问题，显示 Failed to restart。网上的分析是端口有冲突，暂时的解决方法是命令行中输入

> netsh winsock reset

但是重启之后问题依然出现。长期的解决方案是下载 nolsp.exe ，命令行中执行

> noLsp.exe C:\windows\system32\wsl.exe

编码的方案是在宿主系统中编写和维护项目，然后把项目目录挂载到 docker 中运行。docker 运行之后并不需要保存过程文件，所以可以直接删除来维护一个干净的调试环境。假设本地文件夹的位置是 `D:\compiler` ，想要挂载到 `/root/compiler` 位置，使用以下命令：

> docker run -it --rm -v D:\compiler:/root/compiler maxxing/compiler-dev bash

注意 windows 中的目录格式和 Linux 中是不同的，所以挂载时前后的目录斜杠方向相反。

然后在相应的文件夹中放入项目模板，并且在 `git` 中设置自己对应的项目地址。基本的编码环境就设置好了。

Lv1 & Lv2

这部分需要构建一个词法分析和语法分析的框架，前两个lv没有功能的实现，所以只要能够解析代码并且扔掉注释就好了。

`sysy.l` 文件用于进行词法分析，其中定义了每一个最基本的token。词法分析把字节流转换为token流。如果是保留字符，返回相应标识符；如果是数字，识别进制然后转换出对应的数最为返回值；其他的名称作为标识符返回字符串。为了让保留字不被作为标识符返回，需要把他们的定义写在标识符的正则表达式之前。 `sysy.l` 文件基本上已经给出了，只需要补充段注释的正则表达式。段注释由斜杠和星号包裹，且为最短匹配原则，与正则表达式默认的最长匹配不一致，所以需要规定在中间不能出现 `*/` 。也就是说中间可以是没有出现 `*` ，或者 `*` 之后不能紧跟着 `/` 。满足这样形式的字符可以出现多次，所以最终的实现为：

```
BlockComment  "/*"([^\*]|(\*)*[^\*/])*(\*)*"*/"
```

`sysy.y` 定义了语法分析的规则，来得到AST。推导的开始符号是 `CompUnit` ，根据ENBF的规则，需要给出相应的推导规则以及对应的行为。输出中间表示的动作是在解析完整个代码之后才进行的，所以在此处只要把相应的结构关系定义好就行，具体的行为在结构体的对应函数中实现。但是此处的推导规则和ENBF有一些区别，比如此处不能直接定义出现0次或1次，也不能直接定义出现多次。在 `sysy.y` 文件的开头需要给出非终结符的类型定义，每次在之后的推导中定义一个新的非终结符号，就要在此处添加声明。此处的推导规则都在文档里给出了，直接使用就好。

新建一个文件 `AST.h` 用于存放所有 `AST` 的结构。所有的AST都是一个基类 `BaseAST` 的派生类，并在派生类中定义相关的关联结构。这样做的好处是所有的类都可以使用基类的指针来管理，也可以用相同的函数根据派生类实现不同的功能。在 `sysy.y` 进行语法分析的过程中会构建好这些AST的互相关系（也就是填充好对应的内容和指针），最后返回给调用语法分析函数的是一个 `CompUnit` 的指针。对此指针调用输出函数可以根据自定的规则遍历整棵树并输出中间形式。

参考在线文档，定义 `Dump` 函数用于输出。为了方便调试，可以在每个函数的开始设置如果是调试模式就输出对应AST的名称，这样可以看清推导的具体过程。在这个阶段并不需要输出有意义的IR，所以这部分可以先放一放。

中间代码（Koopa IR）到最终汇编的生成过程其实就是再次解析代码并生成另一种代码的过程。[illegible]作，主要是解析高级的

可喜可贺的是，将文本形式的 IR 转换为内存形式的库是已经提供的，所以在最终代码生成的过程中我们只要考虑根据相应的，而不用去像之前一样分析语法词法。当然如果没有这个库的话，中间形式的 IR 直接生成内存形式会比文本形式方便得多。

一个完整中间形式的代码会被解析成一个 koopa_raw_program_t ，通过对应不同精细度的列表拆分到函数、基本块和具体指令。由于 IR 中没有复杂的指令，所有的指令都是可以线性转换成相应汇编的，所以只要顺序遍历这个结构，对相应的模块作相应的输出即可。

koopa_raw_program_t 中有一个全局值的部分和函数的部分，遍历时先获得总长度然后拿指针逐个遍历。此处只有函数的列表且只有一个元素。函数中分成数个基本块，基本块分成具体的语句。虽然此处的基本块只有一个，语句类型也只有返回，但是为了之后方便，还是最好构建出整个完整的解析模式：

```
koopa_raw_program_t raw = koopa_build_raw_program(builder, program);

// 处理 raw program
for (size_t i = 0; i < raw.values.len; ++i)  // 处理全局变量和常量分配
{
  if (value->kind.tag == KOOPA_RVT_GLOBAL_ALLOC)  // 类型只能是全局分配
  {

  }
  else cout<<"ERROR"<<endl;
}
for (size_t i = 0; i < raw.funcs.len; ++i)  // 使用 for 循环遍历函数列表
{
  assert(raw.funcs.kind == KOOPA_RSIK_FUNCTION);
  // 获取当前函数
  koopa_raw_function_t func = (koopa_raw_function_t) raw.funcs.buffer[i];

  for (size_t j = 0; j < func->bbs.len; ++j)
  {
    koopa_raw_basic_block_t bb = (koopa_raw_basic_block_t)func->bbs.buffer[j];
    // 进一步处理当前基本块
    for (size_t k = 0; k < bb->insts.len; ++k)
    {
      koopa_raw_value_t value = (koopa_raw_value_t)bb->insts.buffer[k];
      if (value->kind.tag == KOOPA_RVT_ALLOC){
      }
      else if (value->kind.tag == KOOPA_RVT_LOAD){
      }
      else if (value->kind.tag == KOOPA_RVT_STORE){
      }
      else if (value->kind.tag == KOOPA_RVT_GET_PTR) {
      }
      else if (value->kind.tag == KOOPA_RVT_GET_ELEM_PTR) {
      }
      else if (value->kind.tag == KOOPA_RVT_BINARY){
      }
      else if (value->kind.tag == KOOPA_RVT_BRANCH){
      }
      else if (value->kind.tag == KOOPA_RVT_JUMP){
      }
      else if (value->kind.tag == KOOPA_RVT_CALL){
      }
      else if (value->kind.tag == KOOPA_RVT_RETURN){
      }
    }
  }
}
// 释放 Koopa IR 程序占用的内存
```

```
koopa_delete_raw_program_builder(builder);
```

Lv3

这一章要实现实现一个能够处理表达式 (一元/二元) 的编译器。

IR 生成

IR 的工作主要可以分为两部分： EBNF 对应语法解析规则的修改和输出规则的定义。 在推导规则中不能出现部分结构的或形式，所以遇到 | 时需要将整个结构展开写一次。对于多种推导规则的情况，可以在 AST 中设置一个变量标明是第几种推导得出的结果，以节约事后判断的烦恼。运算的优先级是根据推导顺序定义的，并且推导的结果靠前的部分是与推导的头相同，靠后的部分是下一级运算，这样保证了运算是从左到右进行的。此处的计算规则都很明了，也不存在歧义，所以直接按照语法规范构建即可。在 Dump 函数中，注意运算的规则，应当先算前面部分再算后半部分，也就是先调用前面部分的 Dump ，再调用后半部分的 Dump 。

一元运算在生成中间表示时可以看作与常数进行二元运算。复杂的算术运算经过语法分析已经知道了计算的顺序，调用子一级的运算之后将前后两个结果进行二元运算。这样所有的运算都可以拆成二元运算的复合。然后使所有中间变量变成单赋值的——方法是设定一个全局变量记录使用到的个数，逐个增加编码。注意在每次使用中间变量之后要给全局变量增加1。

为了方便之后的编码，可以把 Dump 函数的返回值设置为 string ，记录保存这部分计算最终结果的临时变量名称，这样在后期加入例如函数多个参数调用的时候方便记录所有的参数。返回一个由被调用函数构建的字符串是很危险的，因为调用过程结束之后对应的结构都被摧毁了。但是以 string 座位返回之时编译器会构造出一个临时对象，保证这种危险的调用其实是没有问题的。

RISC-V 生成

这里的 value->kind.tag 不一定是 KOOPA_RVT_RETURN 了，可能是一个二元运算，也可能是一个整数。此处所有的计算可以都放在寄存器中，所以此时对寄存器的分配和中间变量的分配规则非常相近。唯一的困难在于并不是每一条二元运算都可以直接对应到 RISC-V 汇编，所以有一些命令需要翻译成多条汇编命令来达到同样的效果。构造一个基本块指针到内存相对 sp 位置的索引，这样在用到之前算出来的结果时只要根据基本块的指针找到相应的结果就可以运算了。

```
map<koopa_raw_value_t, int> map;
```

基本块的 tag 为 KOOPA_RVT_BINARY 表示一个二元运算。其两个操作数可能是整数也有可能是之前计算的结果。不论如何，都要把运算数加载到寄存器中，然后根据相应的运算符号作出运算，最后把结果写入到栈上，并且用这个基本块的指针记录相应的位置。

Lv4 & Lv5

IR 生成

这部分内容需要实现变量和常量的定义、赋值和使用，并且应当在适当的位置使用适当的变量/常量。

需要在 EBNF 中添加对应的推导规则。此时遇到了 Bison 不能直接支持的形式——一个部分重复出现若干次。比如这个例子：

> ConstDecl -> "const" BType ConstDef {"," ConstDef} ";"

我的解决方案是把重复的部分看作一个对自己的递归调用的推导规则。比如此处相当于有若干个 ConstDef 的组合，所以把这个语句拆分成两句：

> ConstDef -> B | B "," ConstDef

这里 B 表示原先 ConstDef 的推导规则。也就是说，现在的可重复部分比原来多了一些规则，就是在推导出之前形式的后面再加上一个新的自己。

常量数值在编译时会直接替换为定义的数，所以只要保存在编译器执行的过程当中，需要的时候找到对应的数据替换上。变量是程序可以读取和修改的，所以在最终的程序中有对应的空间储存变量的值。所以在声明过程中遇到常量时，应存储到一个对应关系中，遇到变量 x 时，应当使用 @x = alloc i32 申请一个空间并放好初始化的值。调用这些符号时，应当找到对应作用域的符号来使用。

为了明确一个符号的作用域，在使用时找到对应的符号，要建立一个符号表。符号表的层次关系是由一组组 {} (也就是 Block)定义的，每个 {} 包括了新的一组符号作用域，其内部的符号优先于外层的符号被使用。如果没有则逐层向外寻找声明过的符号。对此，设计符号表也随着 {} 的出现和结束变动。考虑一个符号表的栈，遇到新的 Block 时，向栈中压入一个符号表，在 Block 中声明的变量和常量就放到栈顶的符号表中。当 Block 结束时，将栈顶的符号表删除，这样所有的局部符号都不存在了，不会影响接下来的程序。使用一个符号时，从栈顶向栈底遍历符号表找到第一个匹配的条目。对符号表的栈操作可以放在 Block 对应 AST 的 Dump 函数中。

为了方便，符号表保存所有声明的变量和常量。对于变量，应该存储其对应的名称和存放位置，但是在此时还拿不到具体放在哪个位置，由具名符号(例如上文中的 @x)替代，其保存对应的内存位置。所以为了引用到正确的位置，每个具名符号应当是不一样的，考虑到栈的每一层最多只有一个同名的符号，可用栈的高度作为后缀来表示这个具名符号。综上，变量应当存储的就是 ident 和 ident_[depth] 。对于常量，只要记住对应的值并且能对应正确，所以存储的是 ident 和对应的值。为了区分到底是什么类型，可以增加一个 type 到对应 entry 中。所以最终的 entry 可以是这样的：

```
typedef struct
{
  int type;
  int value;
  std::string str;
} Symbol;
```

考虑到之后会有多个函数，可以为每个函数建立符号表，以及全局的符号表，最终的构建如下：

```
typedef std::map<std::string, Symbol> Symbolmap;

typedef struct func_symbol
{
  int depth;
  std::vector<Symbolmap> smap;
  std::set<std::string> nameset;
  func_symbol()
  {
    depth = 0;
  }
} funcsymbol;

typedef struct
{
  Symbolmap globalsymbol;     // 全局的符号表
  std::map<std::string, std::unique_ptr<func_symbol>> funcsymbolmap;   // 映射到各个函数
} Symboltable;
static Symboltable symbt;
```

为了方便访问栈中的元素，在具体的实现中改为使用 vector 。

一个临时变量上。如果出现在左边，意味着要向这个变量存储数据——添加一个新的函数 `assign` ，用 `store` 命令把相应的临时变量的值写入到这个变量对应的地址位置。

常量的定义是可以使用之前定义的常量的，所以在编译时就要完成这部分计算。对这些计算涉及到的 `AST` 添加一个函数 `Calc` ，直接返回相应表达式的结果。

RISC-V 生成

考虑需要分配的栈帧的大小。其实在一次遍历完一个函数之后我们就知道会用到多少空间了。那么最简单的方式就是先遍历一遍函数但是不输出，只计算用到的内存大小。 `KOOPA_RVT_ALLOC` 的大小根据分配的空间决定， `KOOPA_RVT_ALLOC` 会申请4字节空间， `KOOPA_RVT_GET_PTR` 和 `KOOPA_RVT_GET_ELEM_PTR` 用到4字节存放指针， `KOOPA_RVT_BINARY` 用到4字节存放运算的结果， `KOOPA_RVT_CALL` 用到4字节存放返回值。这样每个栈帧的大小就都可以计算了。计算完成之后为 `prelogue` 和 `epilogue` 留出一些空间，然后向4字节对齐。

基本块的 `tag` 为 `KOOPA_RVT_ALLOC` 表示局部变量分配。基本块会指向一个指针，该指针指向一个整数，一个数组，或者一个指针类型。指针和整数的大小都是4字节，数组的大小需要计算。

```
if (value->kind.tag == KOOPA_RVT_ALLOC){
  // cout<<"tag = "<<value->ty->tag<<endl;
  if (value->ty->tag == KOOPA_RTT_POINTER)
  {
    if (value->ty->data.pointer.base->tag == KOOPA_RTT_INT32)
    {
      map[value] = allc;
      allc = allc + 4;
    }
    else if (value->ty->data.pointer.base->tag == KOOPA_RTT_ARRAY)
    {
      int sz = calc_alloc_size(value->ty->data.pointer.base);
      map[value] = allc;
      allc += sz;
    }
    else if (value->ty->data.pointer.base->tag == KOOPA_RTT_POINTER)
    {
      map[value] = allc;
      allc = allc + 4;
    }
  }
  // cout<<"AFTER ALLC: " << allc<<endl;
}
```

基本块的 `tag` 为 `KOOPA_RVT_LOAD` 表示从指定内存地址读出数据。考虑 `load` 的对象。如果是全局的变量，获取全局变量的地址后从地址读数据，存到内存上。如果是局部变量，其地址相对 `sp` 的位置记录在 `map` 中，以申请该变量时的基本块作为索引。不然，这是一个数组的地址，其地址存放在 `map[value]` 对应的空间上，所以需要两次 `load` 才能获得想要的信息。

```
if (value->kind.tag == KOOPA_RVT_LOAD){
  if (value->kind.data.load.src->kind.tag == KOOPA_RVT_GLOBAL_ALLOC){
    cout << "  la    t0, " << value->kind.data.load.src->name+1 << endl;
    cout << "  lw    t0, 0(t0)" << endl;
    if (allc >= -2048 && allc <=2047)
    {
      cout << "  sw    t0, " << allc << "(sp)" << endl;
    }
    else
    {
      cout << "  li    t1, " << allc << endl;
      cout << "  add   t1, t1, sp" << endl;
      cout << "  sw    t0, 0(t1)" << endl;
```

```
  else if (value->kind.data.load.src->kind.tag == KOOPA_RVT_ALLOC)
  {
    if (map[value->kind.data.load.src] >= -2048 && map[value->kind.data.load.s
    {
      cout << "  lw    t0, " << map[value->kind.data.load.src] << "(sp)" << en
    }
    else
    {
      cout << "  li    t1, " << map[value->kind.data.load.src] << endl;
      cout << "  add   t1, t1, sp" << endl;
      cout << "  lw    t0, 0(t1)" << endl;
    }

    if (allc >= -2048 && allc <=2047)
    {
      cout << "  sw    t0, " << allc << "(sp)" << endl;
    }
    else
    {
      cout << "  li    t1, " << allc << endl;
      cout << "  add   t1, t1, sp" << endl;
      cout << "  sw    t0, 0(t1)" << endl;
    }
    map[value] = allc;
    allc = allc + 4;
  }
  else{
    if (map[value->kind.data.load.src] >= -2048 && map[value->kind.data.load.s
    {
      cout << "  lw    t0, " << map[value->kind.data.load.src] << "(sp)" << en
    }
    else
    {
      cout << "  li    t1, " << map[value->kind.data.load.src] << endl;
      cout << "  add   t1, t1, sp" << endl;
      cout << "  lw    t0, 0(t1)" << endl;
    }

    cout << "  lw    t0, 0(t0)" << endl;

    if (allc >= -2048 && allc <=2047)
    {
      cout << "  sw    t0, " << allc << "(sp)" << endl;
    }
    else
    {
      cout << "  li    t1, " << allc << endl;
      cout << "  add   t1, t1, sp" << endl;
      cout << "  sw    t0, 0(t1)" << endl;
    }

    map[value] = allc;
    allc = allc + 4;
  }
}
```

基本块的 `tag` 为 `KOOPA_RVT_STORE` 表示向指定内存地址写入数据，其可以看作 `load` 的逆过程，原理几乎相同不再赘述。

Lv6 & Lv7

`if-else` 的文法在 `EBNF` 中是有二义性的，会产生经典的悬空 `else` 问题。改写文法的根据是将 `Stmt` 分为 完全匹配（`MS`）和 不完全匹配（`UMS`）两类，并且在 `UMS` 中规定 `else` 右结合。完全匹配的文法是所有无法在其后面跟 `else` 的语句，不完全匹配的文法后面可能可以跟 `else` 。

> Stmt -> MS | UMS
>
> MS -> ';' | Exp ';' | RETURN ';' | RETURN Exp ';' | LVal '=' Exp ';' | Block | IF '(' Exp ')' MS ELSE MS | WHILE '(' Exp ')' MS | BREAK ';' | CONTINUE ';'
> UMS -> IF '(' Exp ')' Stmt | IF '(' Exp ')' MS ELSE UMS | WHILE '(' Exp ')' UMS

`continue` 和 `break` 语句操作的都是最里层的循环结构，设置一个栈保存循环结构 (也就是保存各层循环的命名)，在遇到 `continue` 和 `break` 语句时分别跳转到栈顶循环的末尾或者开头。

这部分要注意的一个点是每个代码块都不能是空的,并且最终一定要以唯一的跳转或者返回作为结尾，这和汇编时的 `label` 不太一样。所以在生成 `IR` 时，应当时刻注意当前是不是处于一个结束的代码块下。

```
void test()
{
  return;
  int a;
}
```

例如以上函数，函数已经返回了，下面却还有命令。如果直接按照原先的解析规则，就会在终结的代码块下继续生成指令。其实这类命令没有机会被执行，可以直接不生成 `IR` 。但是这样做似乎显得对原来的程序有些不忠诚，而且判断何时停止忽略代码也是很大的问题。对这类代码的处理方式是定义一类 `other_[count]` 的代码块。如果终结的代码块下还要生成命令，就新建一个这样的新的 `Block` 来存放这些命令。当然这些命令最终也没有机会被执行到。

设置全局变量 `blockend` 来表示当前代码块是不是终结。在每个可能发生危险的地方先看是不是终结，如果终结了就不能在下面继续生成代码了。能使得代码块终结的类型只有返回和跳转指令，在遇到这些指令时应设置 `blockend` 为 `1` 。

RISC-V 生成

这部分的最终代码基本上没有太大的改变，再次注意二元运算的结果是不是正确。

Lv8

这部分的目的是处理函数和全局变量

IR 生成

函数的引入意味着一个 `CompUnit` 将可以推导出多个函数，当然可能推导出一些全局变量和常量的定义。但是对 `CompUnit` 推导的修改好像不那么简单，所以我采取了一种“间接”的方式——先由 `CompUnit` 推导出一个固定的 `AST` ，再用这个 `AST` 作文章。然后又发现了新的问题——全局变量的声明和函数声明的前两个词是完全一样的，意味着在推导时会遇到“归约-归约冲突”。解决的方案是再用一个“间接”的方式——先推导出表示函数或者全局变量的 `AST` ，再细分到底是哪一种。具体的实现就是：

> CompUnit -> Units
>
> Units -> FunorVar | ConstDecl | CompUnits FunorVar | CompUnits ConstDecl
> FunorVar -> FuncType FuncDef | FuncType VarDef ';'

函数定义较之前多处可选的参数部分。和Lv4中的处理方法一样，拆分非终结符号如下：

> FuncDef -> IDENT '(' ')' Block | IDENT '(' FuncFParams ')' Block
>
> FuncFParams -> FuncFParam | FuncFParam ',' FuncFParams

为了方便目标代码生成部分的处理，在进入被调用函数之后给所有的参数分配空间，并复制到栈上保存。在调用函数时，应当给出所有参数。由于之前已经在实现中把所有的数都保存为一个临时变量后返回对应临时变量的名称，此时只要遍历这个参数的递归定义然后把这些返回的字符串连在一起就是调用的参数。

全局符号不能放在任何一个之前定义的符号表中，应当建立一个新的全局的符号表。这时，找一个符号可能在任何函数的符号表里都找不到，还要再次查找全局符号表来确定符号是否存在(当然在编译的评测中所有的符号都是良好定义的不用考虑没找到的情况)，找一个 `ident` 对应的条目的具体的实现为

```
std::map<std::string, Symbol>::iterator it;
if (currentsymbt == NULL)
{
  it = symbt.globalsymbol.find(ident);
}
else
{
  for (int d = currentsymbt->depth; d >= 0; d--)
  {
    if ((it = currentsymbt->smap[d].find(ident)) != currentsymbt->smap[d].end())
      break;
  }
  if (it == currentsymbt->smap[0].end())
  {
    it = symbt.globalsymbol.find(ident);
  }
}
return it->second.value;
```

应当考虑当前处在全局没有函数符号表的情况，此时访问 `currentsymbt` 指针下的内容会导致段错误。

此处有一个文档没有提到且本地测试样例无法测试的问题——函数调用的跳转是有长度限制的。跳转指令只能跳转到 `PC` 相对位置一定范围内的指令。如果是 `jal` 指令，会有 `20` 位用于立即数，加上最后一位隐含的 `0` 并且考虑有符号数，跳转的相对范围是 `+/- 1M` 。如果是 `jalr` 指令，能用于立即数的长度就更小了，只能跳转到附近 `+/- 2K` 的范围。如果超出此范围，需要把目标函数的地址加载到寄存器中，然后用 `jalr` 指令跳转到这个寄存器的地址。但是此时我们并不知道函数的地址，又怎么能加载到寄存器中呢？就像 `jal` 指令一样，或许存在一种方式，只要把对应的 `label` 写到相应的位置，在链接和加载的时候就会自动补上对应的数值。但是尝试一番，发现 `auipc` 指令的源不能直接使用 `label` 。查阅了大量的资料之后，发现在一篇文章中讲到 `RISC-V Assembler Modifiers` ，可以获取一个 `symbol` 的高位或者低位地址。尤其是这一段，似乎可以帮助解决这个问题：

> The high 20 bits of relative address between pc and symbol. This is usually used with the %pcrel_lo modifier to represent a +/-2GB pc-relative range.

```
label:
  auipc      a0, %pcrel_hi(symbol)    // R_RISCV_PCREL_HI20
  addi       a0, a0, %pcrel_lo(label) // R_RISCV_PCREL_LO12_I
```

假设每个函数都会调用别的函数，所以在开头就保存好ra的值，在函数的末尾返回之前恢复。

基本块的 tag 为 KOOPA_RVT_BRANCH 表示一个分支指令。根据运算的结果选择一个位置继续执行。

```
if (value->kind.tag == KOOPA_RVT_BRANCH){
  if (map[value->kind.data.branch.cond] >= -2048 && map[value->kind.data.branc
  {
    cout << "  lw    t0, " << map[value->kind.data.branch.cond] << "(sp)" << e
  }
  else
  {
    cout << "  li    t0, " << map[value->kind.data.branch.cond] << endl;
    cout << "  add   t0, t0, sp" << endl;
    cout << "  lw    t0, 0(t0)" << endl;
  }

  //cout << "  bnez t0, " << value->kind.data.branch.true_bb->name+1 << endl;
  cout << "  beq   t0, x0, " << local_label << "f" << endl;
  cout << "  jal   x0, " << value->kind.data.branch.true_bb->name+1 << endl;
  cout << local_label++ << ":  jal   x0, " << value->kind.data.branch.false_bb
}
```

此处有一个文档没有提到且本地测试样例无法测试的问题——函数调用的跳转是有长度限制的。跳转指令只能跳转到 PC 相对位置一定范围内的指令。如果是 jal 指令，会有 20 位用于立即数，加上最后一位隐含的 0 并且考虑有符号数，跳转的相对范围是 +/- 1M 。如果是 jalr 指令，能用于立即数的长度就更小了，只能跳转到附近 +/- 2K 的范围。如果超出此范围，需要把目标函数的地址加载到寄存器中，然后用 jalr 指令跳转到这个寄存器的地址。但是此时我们并不知道函数的地址，又怎么能加载到寄存器中呢？就像 jal 指令一样，或许存在一种方式，只要把对应的 label 写到相应的位置，在链接和加载的时候就会自动补上对应的数值。但是尝试一番，发现 auipc 指令的源不能直接使用 label 。查阅了大量的资料之后，发现在一篇文章中讲到 RISC-V Assembler Modifiers ，可以获取一个 symbol 的高位或者低位地址。尤其是这一段，似乎可以帮助解决这个问题：

> The high 20 bits of relative address between pc and symbol. This is usually used with the %pcrel_lo modifier to represent a +/-2GB pc-relative range.

```
label:
  auipc      a0, %pcrel_hi(symbol)    // R_RISCV_PCREL_HI20
  addi       a0, a0, %pcrel_lo(label) // R_RISCV_PCREL_LO12_I
```

但是这里又让我感到迷惑了，这低位地址是 label 为什么到了高位地址又变成 symbol 了？如果理解成 label 是一种 symbol ，在两个位置都写上函数名，却会报错。继续查阅相关的说明文档，后一句的 label 指向的应该是之前载入高地址的语句，并且这个 label 可以用数字表示，引用时加上后缀 b 或者'f'表示定义在该语句之后还是之前。具体的实现方式如下：

```
1:  auipc ra, %pcrel_hi(func_name)
    jalr  ra, %pcrel_lo(1b) (ra)"
```

由于跳转相对地址的计算非常困难，我们可以假设所有的跳转都可能超过立即数的范围，采用这种先载入到寄存器再跳转到寄存器存储的地址的方式。

注意到 B-Type 分支指令也会有立即数范围的限制，并且没有一个相对地址跳转的分支指令。于是可以先让分支指令跳转到附近的位置，再在该位置定义一个长程无条件跳转。

基本块的 tag 为 KOOPA_RVT_CALL 表示一个函数调用。需要准备好所有的参数，如果参数大于8个，多出的参数应该放在栈上传递。在函数调用返回后，保存返回值到内存，建立与该基本块的映

```
  int p1 = value->kind.data.call.args.len > 8 ? 8:value->kind.data.call.args.l
  for (int para = 0; para < p1; ++para)
  {
    if (map[(koopa_raw_value_t)value->kind.data.call.args.buffer[para]] >= -20
    {
      cout<< "  lw   a" << para << ", " << map[(koopa_raw_value_t)value->kind.
    }
    else
    {
      cout << "  li    t0," << map[(koopa_raw_value_t)value->kind.data.call.ar
      cout << "  add   t0, t0, sp" << endl;
      cout << "  lw    a" << para << ", 0(t0)" << endl;
    }

  }
  if (value->kind.data.call.args.len > 8)
  {
    sbrk = 4 * (value->kind.data.call.args.len - 8);
    cout<< "  addi  sp, sp, -" << sbrk << endl;
  }
  if (sbrk)
  {
    for (int para = 8; para < value->kind.data.call.args.len; ++para)
    {
      if (map[(koopa_raw_value_t)value->kind.data.call.args.buffer[para]] >= -
      {
        cout<< "  lw    t0, " << map[(koopa_raw_value_t)value->kind.data.call.
      }
      else
      {
        cout << "  li    t0," << map[(koopa_raw_value_t)value->kind.data.call.
        cout << "  add   t0, t0, sp" << endl;
        cout << "  lw    t0, " << sbrk << "(t0)" << endl;
      }

      cout<< "  sw    t0, " << 4 * (para-8) << "(sp)" << endl;
    }
  }
  cout << local_label << ":  auipc ra, %pcrel_hi(" << value->kind.data.call.ca
  cout << "  jalr  ra, %pcrel_lo(" << local_label++ << "b) (ra)" << endl;
  if (sbrk){
    cout<< "  addi  sp, sp, " << sbrk << endl;
  }
  if (allc >=-2048 && allc <= 2047)
  {
    cout<< "  sw    a0, " << allc << "(sp)" << endl;
  }
  else
  {
    cout << "  li    t0, " << allc <<endl;
    cout << "  add  t0, t0, sp" << endl;
    cout << "  sw    a0, 0(t0)" << endl;
  }
  map[value] = allc;
  allc += 4;
}
```

Lv9

IR 生成

> A -> IDENT B

增加一条：

> A -> IDENT ArrayDef B

`ArrayDef` 表示一组或多组方括号，方括号内的数为可在编译时计算出的常数表达式。

> ArrayDef -> '[' ConstExp ']' | '[' ConstExp ']' ArrayDef

初始化值也要修改成支持对数组的初始化

> ConstInitVal -> ConstExp | '{' '}' | '{' ConstArrayInitVal '}'
>
> ConstArrayInitVal -> ConstInitVal | ConstInitVal ',' ConstArrayInitVal

变量数组的初始化也类似，不过初始化值可以在程序运行时计算出来。

> InitVal -> Exp | '{' '}' | '{' ArrayInitVal '}'
>
> ArrayInitVal -> InitVal | InitVal ',' ArrayInitVal

定义的数组也需要加入符号表，需要记录的相关信息有数组的名称，对应内存的地址(也就是带数字后缀的命名)。数组本身和初始化值不需要保存在符号表，因为初始化的值会直接保存到相应的申请的空间中。为数组相关的 `AST` 定义一个计算维数的函数 `ArrCalc` ，计算每一维的宽度并且输出成嵌套的定义形式。

全局数组不论是变量还是常量都有一样的处理方式——在程序开头申请存储空间并填上初始化值。函数内部的数组不能直接赋值，要把相应的数load到对应的地址。地址的计算使用 `getelemptr` 命令，给出上一层数组的地址和偏移量，可以得到下一层数组的地址。

数组赋值的参数可能是不满的，所以在一个 `{}` 结束之后如果没有填满对应的元素，需要补充对应数量的 `0` 。最终输出的IR应当是所有 `{}` 和数据都补齐全的形式。在遇到空的数组时应当应用最大的对齐到当前位置的空数组，但是注意一个 `{}` 意味着维数要缩小一个单位。为此，定义递归调用的 `fillinit` 函数，记录当前的深度。在当前位置对应的数组大小是对齐的数组大小和当前深度对应的数组大小的较小值。在一个元组结束时如果还没有填满当前的大小，补上相应的0。

数组的运算是逐层解析指针最终得到对应地址的过程。申请高维数组之后返回的是开始地址，使用 `getelemptr` 命令，给出上一层数组的地址和偏移量，可以得到下一层数组的地址。在拿到存储对应元素的地址之后，使用 `load` 或者 `store` 指令操作对应的元素。

然后是数组作为参数的传递以及解析。
数组作为参数传递的是对应的地址，并且这和 `Lval` 对数组元素的操作不一样，需要定义新的函数 `pDump` 。在 `Lval` 推导出的数组中，得到的一定是一个数，所以得到的是存储数的地址。但是在参数传递时传入的应当是存着数组对应位置的地址的地址。可能是编码上的问题，这个地址存放的是我们要的数组的地址的地址，显得比较奇怪。在被调用函数中，第一层的指针没有维度的，使用的方式与之前不同，用 `getptr` 命令取到之后的指针。

RISC-V 生成

这里的工作是处理空间分配时数组的情况，以及对指针的操作。

定义全局数组的分配函数 `global_alloc`

```
void global_alloc(koopa_raw_value_t value)
```

```
    if (t->kind.tag == KOOPA_RVT_INTEGER)
      cout << "  .word " << t->kind.data.integer.value << endl;
    else
      global_alloc(t);
  }
}
```

对当前 aggregate 中的每一个元素递归调用。如果是最后一层，输出对空间的分配和相应的初始化值；否则是中间层，递归调用为此层的每一个元组分配空间。

定义计算数组空间的函数 calc_alloc_size

```
int calc_alloc_size(koopa_raw_type_t value)
{
  if (value->tag == 0)
    return 4;
  else
    return value->data.array.len * calc_alloc_size(value->data.array.base);
}
```

如果对应 value 的 tag 等于 0 ，说明这是一个整数，也就是数组的最后一层的元素，占用 4 字节空间；否则一定还有下一层，总的空间就是该层的长度乘以接下来一层的数组空间。

需要对全局变量作出分配。全局变量储存在 raw.values 中。每一个元素都是 KOOPA_RVT_GLOBAL_ALLOC 的形式。分配的方法有三种： aggregate 也就是赋值的数组，零初始化片段和整数类型，分别作出对应的分配

```
· (size_t i = 0; i < raw.values.len; ++i)

koopa_raw_value_t value = (koopa_raw_value_t) raw.values.buffer[i];
cout << "  .data" << endl;
cout << "  .globl " << value->name+1 << endl;
cout<<value->name+1<<":"<<endl;

if (value->kind.tag == KOOPA_RVT_GLOBAL_ALLOC)
{
  if (value->kind.data.global_alloc.init->kind.tag == KOOPA_RVT_AGGREGATE)
  {
    global_alloc(value->kind.data.global_alloc.init);
  }
  if (value->kind.data.global_alloc.init->kind.tag == KOOPA_RVT_ZERO_INIT)
  {
    cout << "  .zero " << calc_alloc_size(value->kind.data.global_alloc.init->ty)<< endl
  }
  if (value->kind.data.global_alloc.init->kind.tag == KOOPA_RVT_INTEGER)
  {
    cout<<"  .word "<<value->kind.data.global_alloc.init->kind.data.integer.value<<endl;
  }
}
cout<<endl;
```

基本块的 tag 为 KOOPA_RVT_GET_PTR 和 KOOPA_RVT_GET_ELEM_PTR 都表示表示数组地址的计算。如果给出了第一维度，采用 KOOPA_RVT_GET_ELEM_PTR ，否则采用 KOOPA_RVT_GET_PTR 。为了计算偏离的值，需要从对应的结构中追寻指针，找到开始定义数组的位置，调用 calc_alloc_size 计算一个单位的大小。注意对数组地址的操作方式和对数组中数的操作是不同的，所以在最后一层应当作出不一样的处理。

复习一下原先的 EBNF：

```
CompUnit      ::= [CompUnit] (Decl | FuncDef);

Decl          ::= ConstDecl | VarDecl;
ConstDecl     ::= "const" BType ConstDef {"," ConstDef} ";";
BType         ::= "int";
ConstDef      ::= IDENT {"[" ConstExp "]"} "=" ConstInitVal;
ConstInitVal  ::= ConstExp | "{" [ConstInitVal {"," ConstInitVal}] "}";
VarDecl       ::= BType VarDef {"," VarDef} ";";
VarDef        ::= IDENT {"[" ConstExp "]"}
                | IDENT {"[" ConstExp "]"} "=" InitVal;
InitVal       ::= Exp | "{" [InitVal {"," InitVal}] "}";

FuncDef       ::= FuncType IDENT "(" [FuncFParams] ")" Block;
FuncType      ::= "void" | "int";
FuncFParams   ::= FuncFParam {"," FuncFParam};
FuncFParam    ::= BType IDENT ["[" "]" {"[" ConstExp "]"}];

Block         ::= "{" {BlockItem} "}";
BlockItem     ::= Decl | Stmt;
Stmt          ::= LVal "=" Exp ";"
                | [Exp] ";"
                | Block
                | "if" "(" Exp ")" Stmt ["else" Stmt]
                | "while" "(" Exp ")" Stmt
                | "break" ";"
                | "continue" ";"
                | "return" [Exp] ";";

Exp           ::= LOrExp;
LVal          ::= IDENT {"[" Exp "]"};
PrimaryExp    ::= "(" Exp ")" | LVal | Number;
Number        ::= INT_CONST;
UnaryExp      ::= PrimaryExp | IDENT "(" [FuncRParams] ")" | UnaryOp UnaryExp;
UnaryOp       ::= "+" | "-" | "!";
FuncRParams   ::= Exp {"," Exp};
MulExp        ::= UnaryExp | MulExp ("*" | "/" | "%") UnaryExp;
AddExp        ::= MulExp | AddExp ("+" | "-") MulExp;
RelExp        ::= AddExp | RelExp ("<" | ">" | "<=" | ">=") AddExp;
EqExp         ::= RelExp | EqExp ("==" | "!=") RelExp;
LAndExp       ::= EqExp | LAndExp "&&" EqExp;
LOrExp        ::= LAndExp | LOrExp "||" LAndExp;
ConstExp      ::= Exp;
```

调整之后的推导规则：

```
CompUnit          -> Units
Units             -> FunorVar
                   | ConstDecl
                   | CompUnits FunorVar
                   | CompUnits ConstDecl
FunorVar          -> FuncType FuncDef
                   | FuncType VarDef ';'

FuncType          -> INT
                   | VOID
BType             -> INT

FuncDef           -> IDENT '(' ')' Block
                   | IDENT '(' FuncFParams ')' Block
FuncFParams       -> FuncFParam
```

```
Block              -> '{' BlockItem '}'
BlockItem          ->
                    | Stmt BlockItem
                    | Decl BlockItem

Decl               -> ConstDecl
                    | VarDecl
ConstDecl          -> CONST BType ConstDef ';'
ConstDef           -> IDENT '=' ConstInitVal
                    | IDENT ArrayDef '=' ConstInitVal
                    | IDENT '=' ConstInitVal ',' ConstDef
                    | IDENT ArrayDef '=' ConstInitVal ',' ConstDef
ArrayDef           -> '[' ConstExp ']'
                    | '[' ConstExp ']' ArrayDef
ConstInitVal       -> ConstExp
                    | '{' '}'
                    | '{' ConstArrayInitVal '}'
ConstArrayInitVal  -> ConstInitVal
                    | ConstInitVal ',' ConstArrayInitVal
ConstExp           -> Exp
VarDecl            -> BType VarDef ';'
VarDef             -> IDENT
                    | IDENT ArrayDef
                    | IDENT '=' InitVal
                    | IDENT ArrayDef '=' InitVal
                    | IDENT ',' VarDef
                    | IDENT ArrayDef ',' VarDef
                    | IDENT '=' InitVal ',' VarDef
                    | IDENT ArrayDef '=' InitVal ',' VarDef
InitVal            -> InitVal
                    | InitVal ',' ArrayInitVal

Stmt               -> MS
                    | UMS

MS                 -> ';'
                    | Exp ';'
                    | RETURN ';'
                    | RETURN Exp ';'
                    | LVal '=' Exp ';'
                    | Block
                    | IF '(' Exp ')' MS ELSE MS
                    | WHILE '(' Exp ')' MS
                    | BREAK ';'
                    | CONTINUE ';'
UMS                -> WHILE '(' Exp ')' UMS
                    | IF '(' Exp ')' Stmt
                    | IF '(' Exp ')' MS ELSE UMS

Exp                -> LOrExp
LOrExp             -> LAndExp
                    | LOrExp OR LAndExp
LAndExp            -> EqExp
                    | LAndExp AND EqExp
EqExp              -> RelExp
                    | EqExp EQ RelExp
                    | EqExp NE RelExp
RelExp             -> AddExp
                    | RelExp LE AddExp
                    | RelExp GE AddExp
                    | RelExp '<' AddExp
                    | RelExp '>' AddExp
AddExp             -> MulExp
```

```
                    | MulExp '/' UnaryExp
                    | MulExp '%' UnaryExp
UnaryExp            -> PrimaryExp
                    | UnaryOp UnaryExp
                    | IDENT '(' ')'
                    | IDENT '(' FuncRParams ')'
FuncRParams         -> Exp
                    | Exp ',' FuncRParams

UnaryOp             -> '+'
                    | '-'
                    | '!'
PrimaryExp          -> '(' Exp ')'
                    | Number
                    | LVal
LVal                -> IDENT
                    | IDENT ArrayExp
ArrayExp            -> '[' Exp ']'
                    | '[' Exp ']' ArrayExp
Number              -> INT_CONST
```

实验中很多的细节设计是很微妙的，由于对未来改动的不知晓很多结构的设计可扩展性并不好，所以在未来需要推翻重来。所以在编码的过程中，如果能提早对整体的结果有一定了解，可以减少返工的次数。此外，编译器的设计是很严谨的，任何的输入情况都应该被考虑到。如果在相应的阶段草草阅读说明文档，依赖测试样例调程序，会为之后的工作埋下很大的安全隐患。在一个阶段提到一句的问题或许在实现上很复杂(比如立即数范围的问题)，而且当时来看没有影响，但是后期想要再找BUG就要更多的时间。此外，编译器的编写最好能有备忘录，想要实现一个功能之前先在备忘录中把整体的框架都设计完备，细节都标注出来，然后逐个实现函数。不然很容易在钻入函数之后就忘记了与之前的呼应，留下各种隐患。

> 本文使用 Zhihu On VSCode 创作并发布

编辑于 2022-05-11 23:48

编译　北京大学　北京大学信息科学技术学院

文章被以下专栏收录

CSAPP Lab
2020PKU CSAPP课程的Lab解答

北京大学编译实践
SysY -> Koopa IR -> RISCV

推荐阅读

深入理解基于RISC-V ISS Spike的仿真系统：探索...

RISC-V工具链进展 2019/11/18

作者：PLCT实验室 公众号文章地址： RISC-V工具链进展 2019/11/18大家好，后续我们软件所PLCT实验室将通过HelloGCC社区，每周定期介绍一下在RISC-V开源工具链上的进展。希望能够吸...

记录一个小白对于RISC-V的认知过程

一、前言 在真正的去了解RISC-V之前，其实我只是在自己关注的一些嵌入式的公众号上看到过关于它的文章，而且看到过很多次，可见它现在真的很火，但是我都没有仔细的去了解过。最近有幸和几...

发表于数字电路杂...

RISC-V Vector Ex 习笔记

RISC-V的开发人员为 处理器的需求，现在已 向量扩展加入到了RIS 由于还没有发布final 本文中的某些指令，都 被新的版本所更新。

啊哈

4 条评论 切换为时间排序

写下你的评论...

Matrixtang IP 属地浙江 · 05-13
好 支持
1

黑猫警长 IP 属地天津 · 05-12
这个外校的可以写吗，测试公开吗
赞

江枫渔火 (作者) 回复 黑猫警长 IP 属地北京 · 05-13
校外一样可以跟随文档编写并在本地测试 但是无法进行在线测试
赞

黑猫警长 回复 江枫渔火 (作者) IP 属地天津 · 05-13
可以的👍👍
赞