编译原理

MiniDecaf 编译器实验报告 -- STAGE 5

2021010706 岳章乔

一、 思考题

step 11:

1.

在现有的基础上，编译器在翻译函数内数组定义，生成 `MALLOC Ti, size` 这种中间代码，对应的是

```
addi sp, sp -size
mv <Ti>, sp
```

`size` 是立即数。

现在如果大小可变，那么在翻译数组定义的时候，可以把 MALLOC 后面的操作数改为是一个变量，那么就有

```
MALLOC Ti, Tj
```

对应指令

```
imuli <Tj>, <Tj>, -4
add sp, sp, <Tj>
mv <Ti>, sp
```

在改动的时候，需要同步修改前端语法分析阶段，允许在函数域内定义可变长度数组。

step 12:

1.

这个跟数组寻址模式有关系。实际上，对于定长数组

```
T arr[M1][M2]...[Mn]
```

对其元素进行引用，如

```
arr[i1][i2]...[in]
```

其对应地址为

```
arr + sizeof(T) * Mn(M(n-1)(...(M3(M2*i1 + i2) + i3)...) + i(n-1)) +in)
```

其也等价于

$$\text{arr} + \text{sizeof}(T) \cdot \sum_{k=1}^{n} i_k \left(\prod_{j=k+1}^{n} N_j \right)$$

无论是哪种方式求值，其都与 N1 无关，因此不需要记录第一维信息。

在本编译器的实现中，当函数数组参数没有指定第一维的时候，会把第一维填充为 1，以通过构造符号表阶段的类型检查。

二、实验内容

2.0. 约定

2.1. 实验需求

1. 实现数组的定义和存取。
2. 实现数组传参。
3. 实现数组初始化值。

2.2. 具体实现

2.2.1. 词法分析

补全 `frontend/scanner.l` 的单词表。

step-11:

仅需要识别方括号，作为数组下标的识别符。

step-12:

无。

2.2.2. 语法分析

step-11:

原有的 非终结符 `VarDecl` 展开为基本类型变量的定义。

现在对其扩充，可以定义数组类型，具体是增加产生式

```
VarDecl --> Type IDENTIFIER ([INTCONST])^+ ;
```

这里用非终结符 `IndexList` 代表 `([INTCONST])^+`。

如此一来，需要修改 `VarDecl` 的构造函数，记录数组长度，这里配套使用新定义的 `ast::ArrayType` 表示这种层次类型，用 `dimList` 将之记录。可以通过扩展这种方案，实现结构体。

对于数组引用，则修改非终结符 `Lvalue` 增加产生式

```
Lvalue --> IDENTIFIER RankList
```

这里 RankList 表示数组下标。

为此扩充定义 `ArrayType` 和 `ArrayRef` ，在 `makefile` 补充其链接部分。

step-12:

对于数组的初始化值，扩充定义 `VarDecl` ，使之记录初始化列表：

`VarDecl --> Type IDENTIFIER = { InitList };`

这里的 InitList 表示初始化列表，同样也用 step-9 对参数列表的处理方法，定义非终结符 `InitListPrefix`。

对于函数传参，修改其形式参数列表 (`FormalList`) 的定义，

`(Type IDENTIFIER (NIndexList)? COMMA) * Type IDENTIFIER (NIndexList)?`

这里，`NIndexList` 表示的是数组维度，展开为

`NIndexList --> [ (ICONST)? ] IndexList`

IndexList 已于上文定义，这么作的目的是因为寻址模式跟第一维无关，可以省略。为了通过构造符号表阶段的类型检查，内部记录被忽略掉的第一维大小为一。

2.2.3. 语义分析

构造符号表：

修改 `translation/build_sym.cpp`。

step-11:

值需要处理数组定义。

数组类型可以递推获得，具体是

递归起点：

```
res_type = base_type->ATTR(type)
```

这个目前肯定是 Int 。

接着由于 C 语言的数组是行主导，因此反向推出 ArrayType 的类型，反向遍历 `dimList`

```
res_type = new type::ArrayType(res_type, *it)
```

这里 `it` 是 `dimList` 的反向迭代器。

step-12：

只需要处理全局数组的初始值。如果有，则用 `var->setGlobalArrInit` 将之记录。

`setGlobalArrInit` 是自行定义的，仿照基本类型的全局变量的 `setGlobalInit` 。

类型检查：

step-11:

对上文提到的 `ArrayRef` 检查即可。

先确保符号表上存在，而且为数组符号。

接着检查下标列表长度和初始化维度列表是否一致，类型是否为 `Int`，如果一致，而且是整形，则用 `getElementType()` 递推返回类型。

step-12:

检查函数引用的时候，对数组参数作特殊处理：如果形式参数是数组，而且实际参数也是，则也认为是参数类型匹配。

2.2.4. 中间代码生成

这里需要考虑目标代码的生成情况。

step-11:

分数组定义和数组存取两部分讨论。

数组定义又分全局和局部两种情况。

对于局部变量，生成中间代码 `Tac::ALLOC T, size` ，之后对应的是压栈指令。

对于全局变量，扩充 `tac/trans_helper.cpp` ，payload 放置数组大小的 `PADDING` 。

数组存取，思想是先算下标地址，例如

`a[i] == *(a + i)`

实际上，就是先算 `a + i` ，这里的加法是整形加上地址，需要用 `sizeof(a[i])` 处理。

可以分为两步：

第一步，算 `p = a+i`

第二步，算 `*p` ，然后对其存取。

第一步就是上文提到的

$$\text{arr} + \text{sizeof}(T) \cdot \sum_{k=1}^{n} i_k \left(\prod_{j=k+1}^{n} N_j \right)$$

这里构造三地址码节点 `Tac::PTRADD` 对应指针加法。

第二部步和 step-10 类似，分别定义 `Tac::SAVEMEM` （ `SaveValAt` ）和 `Tac::LOADMEM` （ `LoadValAt` ），存取 `*p` 。

而 `a` 的获取方式，是参照 step-10 的 `LoadGSym` (其实是一样的，只是加上了标记，表示取地址而不是取值)，定义 `LoadGAddr` 。

同样的也在 `tac/trans_helper.cpp` 做出相应的配套。

关于数组存取，同样也是修改 `AssignExpr` 和 `LvalueExpr` 的翻译函数。

step-12 :

分定义和读两个部分。

需要修改变量定义和左值表达式。

对于变量定义，如果有初始化列表，

对于局部数组，先调用 `memset` 对数组清零，然后计算偏移量，用类似 step-11 的方法赋初始化值。

对于全局数组，用 `PayLoad` 的 `WORD` 属性表示各个初始化值，剩余空间用 `PADDING` 表示。

对于左值表达式，由于可以传数组，因此对于全局变量，需要提取其地址，而不是地址对应的值，用 `genLoadGAddr` 将之表示。

同样也对新增的三地址节点，构造输出部分，方便在 `-l 3` 阶段查看。

2.2.5. 中间代码优化

编译器对整个数据流图和每一个基本块内的每一条语句，都分别生成其活跃变量集合。

这样的活跃变量分析是后向分析，转移函数为

$$\mathrm{In}[B] = \mathrm{use}[B] \cup (\mathrm{Out}[B] - \mathrm{def}[B])$$

变量 $x \in \mathrm{def}$ ，当且仅当 x 被赋值后才被使用， $x \in \mathrm{use}$ 当且仅当存在对 x 取值的表达式，该表达式在所有对 x 的赋值之前。

对于每一个基本块，其活跃变量分析就是把语句分拆，然后用基本块的方法对每条语句处理其活跃变量。

step-11:

修改 `tac/dataflow.cpp` 的 `computeDefAndLifeUse` ，生成上文提到的 use 和 def 集合。

对于数组符号分配内存，参数从栈顶指针取值，数组相加的结果，从被加的指针和偏移量取值，`LOADMEM` 表示从地址取值，记录其结果的变量，这些都是定值点，更新 DEF 集合；

数组相加使用被加的指针和偏移量，`LOADMEM` 的地址被使用，而 `SAVEMEM` 的源变量和地址都被使用，因此把它们加入到 LiveUse 集合里。

接着对于基本块内每条语句，从后往前计算其 LiveOut 集合，具体是从其后方语句转移：删掉后方被赋值的变量，加入后方的被使用的变量： 具体参考上述对数据流图的做法。

具体而言，就是修改 `tac/dataflow.cpp` 的 `BasicBlock::analyzeLiveness` 函数。

step-12:

无。

2.2.6. 目标代码生成

修改 `asm/riscv_md.cpp`

step-11:

分`SAVEMEM`, `LOADMEM`, `PTRADD` 和 `ALLOC` 四种情况。

`PTRADD` 在这个阶段与普通加法无异，和 `Tac::ADD` 放在一起即可。

`ALLOC T, size` 可以分解为

`addi sp, sp, -size` 和

`mv <T 的寄存器>, sp` 。

`LOADMEM` 按中间代码生成阶段的约定，直接用 `emitunaryTac` 生成即可，如果是 `T1 = *(TO)` ，生成的结果为

`lw <T1>, 0(<T0>)`

对于 `SAVEMEM` ， 如果是 `*(T1) = TO` 则要先用 `genRegForRead` 取 `TO` 和 `T1` ，然后生成

`sw <T0> , 0(<T1>)`

另外，对于 `genLoadGAddr` 阶段生成的带标记的 `LoadGSym`，

在 step-10 的处理方式是

`la <分配的寄存器> , <符号名>` 取得符号的地址，然后用 `lw <分配的寄存器>, 0(<分配的寄存器>)` 将地址的值转移，那就是

```
la <分配的寄存器> , <符号名>
lw <分配的寄存器>, 0(<分配的寄存器>)
```

现在则是没有标记的时候才生成上述指令，有标记的时候表示生成地址，结果为

```
la <分配的寄存器> , <符号名>
```

分别用 `AddInstr` 和 `emitInstr` 生成和输出汇编指令。

step-12:

修改 `emitInstr` 的 `CALL` 的部分，在前文调用 `memset` 的时候，不应带下划线前缀，做一个特判。