

hw9_10实验过程记录

参考资料

参考虎书的9.1节和9.3节。

实验过程

Tile-Pattern

T_stm

1. Label

树型	指令	解释
T_Label(C0)	C0:	block开头

2. Jump

树型	指令	解释
T_Jump(L0)	br label L0	直接跳转

3. Cjump

树型	指令	解释
T_Cjump(op, left, right, t, f) where type=T_int	%L = icmp op i64 left, right br i1 %L, label t, label f	op for eq, ne, slt, sgt, sle, sge
T_Cjump(op, left, right, t, f) where type=T_float	%L = fcmp op double left, right br i1 %L, label t, label f	op for oeq, one, olt, ogt, ole, oge

- 这里的left和right可以是temp或者常量
 - 对于常量，无需再申请新的temp，直接写死在操作数处

4. Move

树型	指令	解释
----	----	----

树型	指令	解释
<code>T_Move(T_Mem(left), right)</code>	<code>%L = inttoptr i64 left to i64*</code> <code>store i64 (double) src, i64* %L</code>	这里 <code>right</code> 是调用 <code>munchExp</code> 后的结果
<code>T_Move(T_Temp(left), T_Temp(right))</code>	<code>left = add (fadd) i64 (double) right, 0</code>	这里要用 <code>AS_Move</code>
<code>src</code> 为其他类型	根据 <code>src</code> 决定对应的指令	目的寄存器指定为 <code>dst</code>

- 首先可以断言的是：Move左侧的exp (即dst) 的类型只可能是Mem或者Temp
 - 如果是Mem，应该用store指令
 - 如果是Temp，再看右侧的exp (即src) 类型决定对应的指令
- 这里对于常量同样无需申请新的temp
- 值得注意的是，如果Move的两个exp都是temp，应该用 `AS_Move` 而非 `AS_Oper`

5. Exp

树型	指令	解释
<code>T_Exp(exp)</code>	根据 <code>exp</code> 决定对应的指令	直接调用 <code>munchExp</code> ，不关心目的寄存器

6. Return

树型	指令
<code>T_Return(exp)</code>	<code>ret i64 (double) exp</code>

T_exp

1. BinOp

树型	指令	解释
<code>T_Binop(op, left, right)</code> where type=T_int	<code>dst = op i64 left, right</code>	<code>op</code> for <code>add</code> , <code>sub</code> , <code>mul</code> , <code>sdiv</code>
<code>T_Binop(op, left, right)</code> where type=T_float	<code>dst = op double left, right</code>	<code>op</code> for <code>fadd</code> , <code>fsub</code> , <code>fmul</code> , <code>fdiv</code>

- 对于left或right是常量的情况，同样直接写死在指令中，无需先放到寄存器里

2. Mem

树型	指令
<code>T_Mem(exp)</code>	<code>%L = inttoptr i64 exp to i64*</code> <code>dst = load i64 (double), i64* %L</code>

- Mem可能对应load或store，而在Move中已经对store的情形做了特判，这里只可能对应load
- 对于exp是常量的情况，写死在inttoptr指令字符串中

3. Temp: 直接返回该temp

4. Name

树型	指令	解释
<code>T_Name(label)</code>	<code>dst = ptrtoint i64* @label to i64</code>	将method name变成地址

5. Const

树型	指令
<code>T_Const(num)</code>	<code>dst = add (fadd) i64 (double) num, 0 (0.0)</code>

6. Call

树型	指令	解释
<code>T_Call(id, obj, args)</code>	<code>%L = inttoptr i64 obj to i64*</code> <code>dst = call ty %L(args)</code>	<code>args</code> 通过调用munchArgs处理

7. ExtCall

树型	指令	解释
<code>malloc(size)</code>	<code>%L = call i64* malloc(i64 size)</code> <code>dst = prttoint i64* %L to i64</code>	<code>size</code> 为常量时直接替换
<code>getint()</code>	<code>dst = call i64 @getint()</code>	
<code>getch()</code>	<code>dst = call i64 @getch()</code>	
<code>getfloat()</code>	<code>dst = call double @getfloat()</code>	
<code>getarray(arr)</code>	<code>%L = inttoptr i64 arr to i64*</code> <code>dst = call i64 @getarray(i64* %L)</code>	<code>size</code> 为常量时直接替换
<code>getfarray(arr)</code>	<code>%L = inttoptr i64 arr to i64*</code> <code>dst = call i64 @getfarray(i64* %L)</code>	<code>size</code> 为常量时直接替换

树型	指令	解释
<code>putint(num)</code>	<code>call void @putint(i64 num)</code>	<code>num</code> 为常量时直接替换
<code>putch(ch)</code>	<code>call void @putch(i64 ch)</code>	<code>ch</code> 为常量时直接替换
<code>putfloat(num)</code>	<code>call void @putfloat(double num)</code>	<code>num</code> 为常量时直接替换
<code>putarray(len, arr)</code>	<code>%L = inttoptr i64 arr to i64*</code> <code>call void @putarray(i64 len, i64* %L)</code>	<code>len</code> 为常量时直接替换
<code>putfarray(len, arr)</code>	<code>%L = inttoptr i64 arr to i64*</code> <code>call void @putfarray(i64 len, i64* %L)</code>	<code>len</code> 为常量时直接替换
<code>starttime()</code>	<code>call void @starttime()</code>	
<code>stoptime()</code>	<code>call void @stoptime()</code>	

代码实现

数据结构

将munchExp的返回值用expres表示，即结果可以是temp或const：

```

1  typedef struct expres_ *expres;
2  struct expres_ {
3      enum { res_const, res_temp } kind;
4      T_type type;
5      union {
6          int i;
7          double f;
8          Temp_temp t;
9      } u;
10 };

```

这样做的目的是：

1. 在处理ConstExp时，可以直接返回常数值
2. 当llvm指令的操作数对应的exp为常数时，可以直接写死在指令中，无需再申请一个temp中转一下
3. 让代码更简单：
 - 直接判断expres是否是常数即可，减少了对exp类型的特判
 - 后续增加优化更容易：比如要增加对T_Cast(floatConst, T_int)的优化，直接返回IntConstRes(num)即可

代码组织

```
1 static void munchStm(T_stm s);
2 static void munchLabelStm(T_stm s);
3 static void munchJumpStm(T_stm s);
4 static void munchCjumpStm(T_stm s);
5 static void munchMoveStm(T_stm s);
6 static void munchExpStm(T_stm s);
7 static void munchReturnStm(T_stm s);
8
9 static expres munchExp(T_exp e, Temp_temp dst);
10 static expres munchBinOpExp(T_exp e, Temp_temp dst);
11 static expres munchMemExp_load(T_exp e, Temp_temp dst);
12 static expres munchTempExp(T_exp e);
13 static expres munchNameExp(T_exp e, Temp_temp dst);
14 static expres munchConstExp(T_exp e, Temp_temp dst);
15 static expres munchCallExp(T_exp e, Temp_temp dst);
16 static expres munchExtCallExp(T_exp e, Temp_temp dst);
17 static Temp_tempList munchArgs(T_expList args, string argsStr, int initNo);
18 static expres munchCastExp(T_exp e, Temp_temp dst);
```

1. munchExp的参数dst

- 目的：在处理MoveStm时，对于 `T_Move(T_Temp(dst), src)` (其中right不是temp)的pattern，可以在递归调用munchExp处理src时直接传入dst，这样不用新申请一个temp中转一下
- 当传入的dst为空时，对应的munchxxxExp函数会自行申请一个新的temp作为dst，并返回

2. munchArgs

- 作用：递归处理call的参数列表args，返回对应的temp列表，同时指令对应的字符串也会同步到传入的argsStr中
- 类似地，如果某个参数是常数，无需加入temp列表，直接写死在字符串中
- 参数：
 - initNo是指llvm指令的源temp占位符起始序号，对于call来说是1 (函数地址占据 `%s0`)
 - argsStr是指llvm指令的参数列表字符串，传入的初始值永远为空字符串 `""`
- 注意到call总有第一个参数this，所以返回的Temp_tempList不会为空

3. MemExp

- 出现在MoveStm的左侧：在munchMoveStm中作特判，输出store指令
- 出现在其他地方：应该输出load指令，通过munchMemExp_load返回得到的数据

4. munchExp的结果总是数字 (i64/double)，所以：

- 在munchxxxExp时，如果得到的是指针i64*，需要在最后加一个ptrtoint指令
- 如果需要的munchExp结果是指针i64*，需要先加一个inttoptr指令

遇到的问题

1. temp重复使用

- 预期：在做指令选择时，新申请的temp的数字应该从irp的最大temp数字加一开始
- 修复 (助教给出)：在读入irp中的temp时，更新temps计数

```
1  if(name >= temps)
2      temps = name + 1;
```

2. Cjump Stm和Binop Exp的左右两个操作数类型不匹配

- 需要在munchxxx函数中做复杂的判断，如果不匹配要加cast
- 改进 (助教给出)：直接在tigerirp.c中判断，若Cjump或Binop中两个Exp的类型不匹配，则加一层T_Cast

测试结果

运行 `make test`，结果符合要求。

开发过程

git提交记录如下：(部分)

hw9_10: fix bug in cjump	9 May 2024 19:49
hw9_10: merge master	9 May 2024 19:34
hw9_10: add llvm code	9 May 2024 08:51
hw9_10: add test-run	9 May 2024 08:31
hw9_10: merge master	9 May 2024 08:30
hw9_10: fix make clean	8 May 2024 17:26
hw9_10: refactor extcall malloc	8 May 2024 13:10
hw9_10: fix call ret type	8 May 2024 11:12
hw9_10: merge master	7 May 2024 23:32
hw9_10: add test-run	7 May 2024 19:21
hw9_10: fix extcall ptr	7 May 2024 19:21
hw9_10: change ptr to i64*	6 May 2024 21:12
hw9_10: add external test	6 May 2024 20:26
hw9_10: merge master	6 May 2024 15:49
hw9_10: merge master	6 May 2024 11:21
hw9_10: refactor and add expres	6 May 2024 10:41
hw9_10: remove cjump and binop's duplicate codes	5 May 2024 20:28
hw9_10: fix AS_Move bug	5 May 2024 19:37
hw9_10: fix munchArgs bug	5 May 2024 16:18
hw9_10: merge master	5 May 2024 14:16
hw9_10: merge master	5 May 2024 14:12
hw9_10: remove duplicate codes	3 May 2024 10:22
hw9_10: fix malloc ptrtoint	2 May 2024 23:54
hw9_10: fix relop for float (oeq, ...)	2 May 2024 23:19
hw9_10: fix temp	2 May 2024 22:33
hw9_10: add debug output in llvmgen	2 May 2024 22:32
hw9_10: complete exp	2 May 2024 21:01
hw9_10: merge master	2 May 2024 18:03
hw9_10: complete part of exp	2 May 2024 17:30