FDMJ 编译器的设计与实现

编译（H）期末项目报告

1 引言

该报告介绍了一个 FDMJ 语言编译器的设计、实现和使用方法，按顺序分别阐述了其前端和后端的原理与实现。编译的整体流程为：

1. 前端：parser 解析源码生成 AST；semant 模块对 AST 进行语义检查（定义-使用关系、类继承关系、类型正确性）并通过 translate 模块生成 Tiger IR；Tiger IR canonical 化（线性化、建立基本块）。
2. 后端：将 Tiger IR 直译为非 SSA 的 LLVM IR（不可运行）；LLVM IR 的 SSA 化；SSA LLVM IR 翻译为使用虚拟寄存器（temp）的 ARM 汇编；对 ARM 汇编进行活动分析；寄存器分配。

本文将按顺序分别详述上面各个阶段。为了便于理解，本文使用一个 FDMJ 的浮点数冒泡排序程序为例，说明各个阶段如何对具体的程序生效。最后，本文将说明如何构建和使用本编译器。

FDMJ 浮点数冒泡排序程序如下：

```
public int main() {
  int i=0;
  float[] a = {6, 3, 0, 5, 9, 1, 2};
  class b1 bo;
  bo=new b1();
  bo.bubbleSort(a, length(a));
  while (i < length(a)) {
    putnum(a[i]);
    putch(32);
    i=i+1;
  }
  putch(10);
  return 0;
}

public class b1 {
  public int bubbleSort(float[] array, int size) {
```

```
    int i=0;
    float temp;
    if (size <= 1) {
      return 0;
    }
    while (i < size - 1) {
      if (array[i] > array[i + 1]) {
        temp = array[i];
        array[i] = array[i + 1];
        array[i + 1] = temp;
      }
      i=i+1;
    }
    return this.bubbleSort(array, size - 1);
  }
}
```

2 Parsing

FDMJ 有这些 token：注释符号、标识符、数字、比较运算符、算术运算符、成员运算符（.）、圆/方/花括号、关键字。lex 中，匹配到标识符会返回 `A_IdExp (A_pos (...), String (yytext))`，匹配到各类字面值，(整数、浮点、布尔）会返回对应的 `A_<Type>Const (A_pos (...), <yytext to value>)`，匹配到其他 token 时则返回其位置，即 `A_pos (line, col)`。

yacc 中会接受 lex 传过来的 token 序列然后匹配到 FDMJ 的语法。对应语法的动作是：构造并返回对应于该语法的 AST 节点。以示例程序的第 3 行为例：

```
float[] a = {6, 3, 0, 5, 9, 1, 2};
```

会被 lex 解析为如下 token 序列

```
FLOAT '[' ']' ID '=' '{' NUM ',' NUM ',' NUM ',' NUM
                         ',' NUM ',' NUM ',' NUM '}' ';'
```

然后匹配到语法规则

```
FLOAT '[' ']' ID '=' '{' CONSTLIST '}' ';' {
  $$ = A_VarDecl($1, A_Type($1, A_floatArrType, NULL), $4->u.v, $7);
}
```

返回值是一个声明了名字为 ID 的字符串内容、类型为浮点数组、初始值为匹配到的常数序列的 AST 节点。

为了检查语法错误，yacc 中还添加了这样的规则，例如：

```
INT ID '=' CONST  {
  fprintf(stderr, "line %d ", $1->line);
  yyerror("expected \';\'\n");
```

```
  exit(0);
}
```

它会检测到末尾没有分号的声明语句，然后报告相应错误。因为 yacc 是顺序匹配的，所以只要在这个错误语法规则之前的 `INT ID '=' CONST ';'` 规则被匹配到，程序就不会报错，因此不影响正确程序通过 parsing。

3 语义检查

3.1 内容

语义检查包括了类型检查、定义-使用检查、类成员检查和类层次关系检查。共需要遍历 3 次 AST，任务和完成之的函数分别为：

1. `I_rec<AstNodeKind>`：记录类名字，类继承关系及各个类中含有的成员变量、方法的签名。在最外层 `I_recClassDeclList` 结束后，会创建好一个存放了所有类信息的表格 `cenv`，每个 `cenv` 表项 class entry 都包括了该类声明的 AST 节点，父类名，状态（用于检查 循环继承），`vtbl`（成员变量表）和 `mtbl`（成员方法表）。其中，
   - `vtbl`：表项 var entry 包含了变量定义的 AST 节点，变量类型和为其分配的 `Temp_temp`（在此处可以看作为其分配的内部变量名）。
   - `mtbl`：表项 method entry 包含了方法定义的 AST 节点，该方法所属的类名，返回值类型和参数列表。参数列表的每一项都是参数名和类型。简单地说，`mtbl` 的每一项都是一个方法的定义节点和签名。
2. `I_chk<AstNodeKind>`:检查继承关系和父类成员 override。`I_chkClassDecl` 会检查循环继承，方法是这样的：刚创建时每个类在 `cenv` 中的表项，其 `status` 都是 `E_transInit`。每当检查到一个类，如果其 `status` 是 `E_transInit`，就把它设置为 `E_transFind`，然后检查其父类（从 `cenv` 表项中获得）。如果没有父类，则检查结束，按照反方向把这条继承路径上所有表项的 `status` 设置为 `E_transFill`，以后检查时再遇到 `E_transFill` 的类，就可以直接认为检查成功。如果遇到 `E_transFind` 的类，则说明它被第一次遇到之后，检查过程又一次遇到了它，这意味着循环继承，语义检查报错。
3. `transA_<AstNodeKind>`：检查类型、变量定义-使用顺序、类成员、左右值。具体有这些检查：
   i. 赋值的左右两边是否匹配、函数调用实参是否匹配签名、函数返回的表达式类型是否符合签名。`int` 和 `float` 算做匹配，`int` 和 `float` 数组不算匹配，子类对象可以赋值给父类变量。
   ii. 变量使用前有没有定义。过程中，用一个 `venv` 表格记录已经定义的变量和其类型。
   iii. 使用一个类对象的成员时，这个成员是否真的属于这个类。通过检查这个类的 `vtbl` 和 `mtbl` 来得知。
   iv. 被赋值变量是否是左值。只有局部变量、类成员变量、数组成员引用（数组对象本身不可以）是左值。

3.2 重要代码解释

3.2.1 左右值

实现中，唯一可能产生左值表达式的 transA_Exp 函数的返回类型是

```
typedef struct Sem_tyCatTr_ *Sem_tyCatTr;
typedef enum Sem_valCat_ Sem_valCat;
enum Sem_valCat_ { Sem_L, Sem_R };
struct Sem_tyCatTr_ { Ty_ty type; Sem_valCat cat; Tr_exp tr; };
```

Sem_valCat 即表达式的值类别：左值或者右值。

3.2.2 遍历 S_table

```
for (S_Symbol top = tab->top; top != NULL;
     top = ((binder)S_getBinder (tab, top))->prevtop)
  {
    /* top will traverse all keys in tab */
  }
```

4 翻译到 Tiger IR

我的实现中，translate 模块是完全被 semant 模块使用的。在每个 transA_<AstNodeKind> 末尾，检查确认语义正确后，调用翻译函数返回对应于该 AST 节点的 Tiger IR 节点。

实现上有如下要点：

统一对象记录（Unified Object Record，UOR）：在本编译器中，为了实现简便，使用了 UOR 技术。在翻译 A_ClassDeclList 时，每遇到任何一个类中的一个对象，就为其分配一个单独的偏移量。所有的对象都有同样的内存大小，不同类型对象的区别只是在同样大小的内存块中，其成员位于的偏移量不同。为了建立 UOR，在类层次检查之后、全局语义检查之前，需要再插入一次 AST 遍历，使用 I_uor<AstNodeKind> 建立 UOR。

跳转语句和逻辑运算：逻辑运算是被翻译成跳转语句和赋值，例如，

```
1 public int main() {
2   int x;
3   x = 1 || 0;
4   putnum(x);
5   return 0;
6 }
```

就被翻译成

```
  Cjump T_ne, 1:int, 0:int, L1, L0
L0:
  Cjump T_ne, 0:int, 0:int, L1, L2
```

```
L1:
  Move t101:int, 1:int
  Jump L3
L3:
  Move t100:int, t101:int
  putint t100:int:int
  Return 0:int
```

为了实现这一点，使用了 patchList，如果翻译这个节点时不能确定其分支，则用 patchList 填充，待完成其分支的翻译后，再 doPatch 进行补完。

下面以 1 节的冒泡排序为例，说明一下语义检查和翻译到 Tiger IR 过程中几个重要的数据结构：

cenv：

class name	class entry
b1	cent1 = E_ClassEntry (cd, S_Symbol ("b1"), E_transInit, vtbl, mtbl)

cent1->vtbl：

var name	var entry
empty	

ent1->mtbl：

method name	method entry
bubbleSort	ment1 = E_MethodEntry (md, S_Symbol ("b1"), Ty_Int (), Ty_FieldList (Ty_Field (S_Symbol ("array"), Ty_Array (Ty_Float ())), Ty_FieldList (Ty_Field (S_Symbol ("size"), Ty_Int())))))

UOR：

member name	member offset
bubbleSort	0

5 Canonical 化

canonical 化就是要把 Tiger IR 中的所有 ESEQ、CJUMP 和嵌套的 CALL 转化为线性的 SEQ，进而可以变成 `T_stmList`，然后根据控制流划分为基本块。

删除 ESEQ，就是把其中的 SEQ 语句提出来先完成，然后把最后的 EXP 放到最后，作为计算值的表达式。划分基本块是按照标签，每个标签后，直到下一个标签或者程序末尾构成一个基本块。每个条件分支对应的 false 跳转目标基本块就直接放在该分支指令后。下面是冒泡排序的一部分 Tiger IR 节点和 canonical 化之后对应的语句：

```
T_Eseq(
  T_Seq(
    T_Move(
      T_Temp(Temp_namedtemp(109,T_int))/*T_int*/,
      T_ExtCall(String("malloc"),
        T_ExpList(
          T_IntConst(4)/*T_int*/,
          NULL)
      ,T_int)/*T_int*/
    ),
    T_Move(
      T_Mem(
        T_Binop(T_plus,
          T_Temp(Temp_namedtemp(109,T_int))/*T_int*/,
          T_IntConst(0)/*T_int*/
        )/*T_int*/
      , T_int)/*T_int*/,
      T_Name(Temp_namedlabel(String("b1$bubbleSort")))/*T_int*/
    )
  ),
  T_Temp(Temp_namedtemp(109,T_int))/*T_int*/
)/*T_int*/
```

```
Move t109:int, malloc 4:int:int
Move Mem(Binop(T_plus, t109:int, 0:int):int):int,  b1$bubbleSort:int
```

后续取值就是 `t109`。

6 瓦片与 LLVM IR 指令选择

6.1 瓦片设计

瓦片形状写在注释中。每个瓦片中，被 <> 包裹的部分不是该瓦片的一部分，在 munch 时不会被吃掉；未被包裹的是该瓦片的一部分，munch 匹配到时会被吃掉。

```
typedef enum TL_tileKind_
{
  /*********
   * T_stm *
   *********/
  /* T_Label (label) */
  TL_stmLabel,
  /* T_Jump (labelJDest) */
  TL_stmUncondBranch,
  /* T_Cjump (cnd, [T_Temp (labelL)/T_[Int/Float]Const (numL)],
   *          [T_Temp (labelR)/T_[Int/Float]Const (numR)],
   *          labelTrueJDest, labelFalseJDest) */
  TL_stmCondBranchTempConst,
  /* T_Cjump (cnd, <expLeft>, <expRight>, labelTrueJDest,
              labelFalseJDest) */
  TL_stmCondBranchOther,
  /* T_Move (T_Temp (temp), T_ExtCall ("malloc", <expSize>, type)) */
  TL_stmMoveMallocTemp,
  /* T_Move (T_Temp (tempDest), T_Mem (<expAddr>, type)) */
  TL_stmMoveLoad,
  /* T_Move (T_Mem (<expAddr>, type), <expSrc>) */
  TL_stmMoveStore,
  /* T_Move (T_Temp (labelDest),
   *         [T_Temp (labelSrc)/T_[Int/Float]Const (numSrc)]) */
  TL_stmMoveTempConst,
  /* T_Move (T_Temp (tempDest), <expSrc>) */
  TL_stmMoveToTemp,
  /* T_Exp (<exp>) */
  TL_stmTExp,
  /* T_Return (<exp>) */
  TL_stmRet,
  /*********
   * T_exp *
   *********/
  /* T_Binop (oper, [T_Temp (tempL)/T_[Int/Float]Const (numL)],
   *          [T_Temp (tempR)/T_[Int/Float]Const (numR)]) */
  TL_expBinopTempConst,
  /* T_Binop (oper, <expL>, <expR>) */
  TL_expBinopOther,
  /* T_Mem ([T_Temp (tempAddr)/T_IntConst (numAddr)], type) */
  TL_expMemTempConst,
  /* T_Mem (<expAdr>, type) */
  TL_expMemOther,
  /* T_Temp (temp) */
  TL_expTemp,
  /* T_Name (label) */
```

```
  TL_expName,
  /* T_IntConst (i) */
  TL_expIntConst,
  /* T_FloatConst (f) */
  TL_expFloatConst,
  /* T_Call (strFuncName, <expFunc>, <expListArgs>, typeRet) */
  TL_expCall,
  /* T_ExtCall (strExtFuncName, <expListArgs>, type) */
  TL_expExtCall,
  /* T_Cast (<expOrigin>, type) */
  TL_expCast,
  /* number of substantial items in this enum */
  TL_tileKindLENGTH,
} TL_tileKind;
```

6.2 瓦片匹配

函数 `TL_tileKind TL_matchTile (TL_stmExp stmexp);` 接受一个可以是 `T_exp` 也可以是 `T_stm` 的 Tiger IR 节点，对其及其子节点进行匹配后，返回匹配到的最大瓦片 `TL_tileKind`。

对于 `T_stm` 和 `T_exp` 节点，分别用 `munchStm` 和 `munchExp` 进行瓦片匹配和指令选择。

举例说明，1 节程序的第 5 行，为 bo 分配堆空间的 Tiger IR 节点为

```
T_Move(
  T_Temp(Temp_namedtemp(109,T_int))/*T_int*/,
  T_ExtCall(String("malloc"),
    T_ExpList(
      T_IntConst(8)/*T_int*/,
      NULL)
  ,T_int)/*T_int*/
)
```

匹配到如下瓦片：

```
/* T_Move (T_Temp (temp), T_ExtCall ("malloc", <expSize>, type)) */
TL_stmMoveMallocTemp
```

在 `munchStm` 中对应该瓦片的动作为

```
/* T_Move (T_Temp (temp), T_ExtCall ("malloc", <expSize>,
 *                                    type)) */
/* <expSize>, the first argument for calling malloc */
subtmp1 = munchExp (stm->u.MOVE.src->u.ExtCALL.args->head);
/* This subtmp2 is for receiving the i64* type return value of
 * malloc */
midtmp1 = Temp_newtemp (T_int);
```

```
sprintf (asmStr1, "%%`d0 = call i64* @malloc(i64 %%`s0)");
emit (LL_Oper (asmStr1, Temp_TempList (midtmp1, NULL),
               Temp_TempList (subtmp1, NULL), NULL));
sprintf (asmStr2, "%%`d0 = ptrtoint i64* %%`s0 to i64");
emit (LL_Oper (asmStr2, Temp_TempList (stm->u.MOVE.dst->u.TEMP,
                                       NULL),
               Temp_TempList (midtmp1, NULL), NULL));
```

生成的 LLVM IR 的对应部分就是

```
%r146 = call i64* @malloc(i64 %r145)
%r109 = ptrtoint i64* %r146 to i64
```

6.3 函数的 prolog 和 epilog

`LL_instrList llvmprolog (string methodname, Temp_tempList args, T_type rettype)` 生成具有 `methodname` 名称、`args` 参数列表和 `rettype` 返回类型的 Tiger IR 函数对应的 LLVM IR 函数的 prolog。prolog 就是原样写出函数名、参数名即类型和返回类型，只是以 LLVM IR 的形式。对于 Tiger IR 的冒泡排序函数 `b1$bubbleSort`，

```
T_FuncDecl(String("b1$bubbleSort"),
  Temp_TempList(Temp_namedtemp(99,T_int),
    Temp_TempList(Temp_namedtemp(100,T_int),
      Temp_TempList(Temp_namedtemp(101,T_int),
        NULL))),
  /* Function body */
)
```

它翻译成的 LLVM IR 的函数 prolog 就是

```
define i64 @b1$bubbleSort(i64 %r99, i64 %r100, i64 %r101) {
```

epilog 就是简单的右花括号 `}` 来封闭函数开头的 `{`。

6.4 指令的目标操作数、源操作数和跳转目标

目标操作数，即 `LL_Oper ()` 的 `d` 参数，虽然是 `Temp_tempList` 类型但是事实上只有一个，就是要被赋值的那个 `Temp_temp`，对应于 Tiger IR 中 `T_Move` 节点左边的表达式。源操作数，即 `LL_Oper ()` 的 `s` 参数，就是要参与运算得到值的若干 `Temp_temp`。比如

```
Move t107:int, Binop(T_plus, t110:int, 8:int):int
```

对应到 LLVM IR，就是

```
%r113 = add i64 %r110, 8
%r107 = add i64 %r113, 0
```

`%r110` 就成了第一行的 src。

跳转目标，即 LL_Oper () 的 j: LL_targets 参数，就是指令可能会跳转到的标签。对于 br label 无条件跳转，只有一个；对于条件跳转会有两个。

7 静态单赋值形式

7.1 简介

静态单赋值（Static Single Assignment，SSA）形式，就是从程序文本角度看，任何一个 Temp_temp 都只作为过一次 dst，即只有一次 def（但运行时不一定，比如循环语句中的赋值，所以说是 static）。

phi 函数是这样的形式：

```
phi <type> [<temp_1>, <block_1>], ..., [<temp_n>, <block_n>]
```

<block_i> 是 <temp_i> 所在块的标签。控制流从 <block_i> 来到 phi 函数，phi 函数的取值就是 <temp_i>。<temp_i> 必须真的定义在 <block_i> 内。

7.2 SSA 化算法

SSA 化算法分为这几步：

- 计算支配节点集合 SSA_computeDomSet；
- 计算立即支配节点 SSA_computeIdom；
- 计算支配边界 SSA_computeDF；
- 插入 phi 函数 SSA_insertPhi；
- Temp_temp 重命名 SSA_rename。

使用如下数据结构存储变量和块图节点的信息。以下分别是变量信息、把 temp 的 num 映射到对应变量信息的哈希表和块图节点的信息。

```
typedef struct SSA_varInfo_ *SSA_varInfo_t;
struct SSA_varInfo_
{
  SET_set_t defsites;
  STK_stack_t stack;
  Temp_temp tmp;
};

/* Hash table using integer as key, not pointers */
typedef struct SSA_tabEnt_ *SSA_tabEnt_t;
struct SSA_tabEnt_
{
  int key;
  SSA_varInfo_t value;
  struct SSA_tabEnt_ *next;
};
typedef struct SSA_intVarInfoTab_ *SSA_intVarInfoTab_t;
```

```
struct SSA_intVarInfoTab_
{
  int size;
  SSA_tabEnt_t *table;
};

typedef struct SSA_bgNodeInfo_ *SSA_bgNodeInfo_t;
struct SSA_bgNodeInfo_
{
  Temp_tempList in; /* in-nodes */
  Temp_tempList Aorig;
  SSA_intVarInfoTab_t Aphi;
  G_node bgNode; /* the node itself */
  SET_set_t domSet;
  G_node idom;
  G_nodeList children;
  SET_set_t df;
};
```

插入 phi 的算法如下:

```
insertPhi () =
  for each bg node n do
    for each temp t in bgNodeInfos[n]->Aorig do
      tInfo = SSA_tabLook (varInfoTab, t->num)
      if tInfo == NULL then
        SSA_tabEnter (varInfoTab, t->num, VarInfo (# of bg nodes, t))
      end if
    end for
  end for

  for each temp t do
    W = defsites (t)
    while W not empty do
      remove node n from W
      for each Y in df[n] do
        if Y not in Aphi and a->key in bgNodeInfos[Y]->in then
          insert "a = phi [a, label(Y)], ..." at the top of Y
          enter a into bgNodeInfos[Y]->Aphi
        end if
        if Y not in its Aorig then
          W[Y] = 1
        end if
      end for
    end while
  end for
```

冒泡排序的 main 函数 block graph 如下：

```
------Basic Block Graph of main------
C1 (0): 1
L7 (1): 2 3
L8 (2):
L9 (3): 1
```

非 ssa 的 main 函数中有如下一段

```
C1:
  %r105 = add i64 0, 0
  ...
L7:
  %r158 = mul i64 -1, 8
  %r159 = add i64 %r106, %r158
  %r161 = inttoptr i64 %r159 to i64*
  %r160 = load i64, i64* %r161
  %r162 = icmp slt i64 %r105, %r160
  br i1 %r162, label %L9, label %L8
...
L9:
  ...
  %r105 = add i64 %r170, 0
  br label %L7
```

这样一来，L7 中的 %r105 就可能来自 C1 或 L9，在这里就需要一个 %r105_1 = phi i64 [%r105_1, C1], [%r105_2, L9]。经过我的编译器的 SSA 化之后，上述代码变成

```
C1:
  %r212 = add i64 0, 0
  ...
L7:
  %r265 = phi i64 [%r277, %L9], [%r212,%C1]
  %r266 = mul i64 -1, 8
  %r267 = add i64 %r249, %r266
  %r268 = inttoptr i64 %r267 to i64*
  %r269 = load i64, i64* %r268
  %r270 = icmp slt i64 %r265, %r269
  br i1 %r270, label %L9, label %L8
...
L9:
  ...
  %r277 = add i64 %r276, 0
  br label %L7
```

可以看到，%r265 代替了原来 L7 开头处的 %r105，并且其赋值变成了来源于 C1 或 L9 的变量的 phi 函数。

8 ARM 汇编生成

LLVM IR 的指令抽象程度高于 ARM 汇编。因为还要做字符串解析（LL_instr 里面没有存储立即数等信息，必须解析字符串），所以方便起见，一条 LLVM IR 指令算作一个瓦片，一一翻译到汇编。过程中，值得注意的有如下三点。

立即数存入寄存器。ARM 汇编中的立即数最多只能 1023，所以要把任意立即数移动到寄存器，就必须分别移动高字节和低字节。把一个 int 类型立即数存入一个新的 Temp_temp 的方法如下：

```
typedef union ARM_immfmt ARM_immfmt_t;
union ARM_immfmt
{
  uint32_t i;
  float f;
  struct
  {
    uint16_t lo, hi;
  } parts;
};

static Temp_temp
intImmIntoTemp (int i)
{
  Temp_temp tmp = TNWT (T_int);
  ARM_immfmt_t fmt = int2fmt (i);
  char *mvlostr = calloc (INSTRLEN, sizeof (char));
  sprintf (mvlostr, "movw `d0, #%d", fmt.parts.lo);
  emit (AS_Oper (mvlostr, TL (tmp, NULL), NULL, NULL));
  if (fmt.parts.hi != 0)
    {
      char *mvhistr = calloc (INSTRLEN, sizeof (char));
      sprintf (mvhistr, "movt `d0, #%d", fmt.parts.hi);
      emit (AS_Oper (mvhistr, TL (tmp, NULL), NULL, NULL));
    }
  return tmp;
}
```

ARM_immfmt_t 的原理是，uint32_t 会按照二进制格式原封不动地存储 int2fmt (int i) 的参数 i，不论正负。parts 中，lo 存储低 16 位，hi 存储高 16 位（根据 GCC 的内存布局），于是分别把这两个 16 位移动到某一 Temp_temp 即可。浮点立即数的处理是类似的。

去除 phi 函数。去除 phi 函数就是插入 phi 函数的逆过程。对于每个 %rx = phi <type> ..., [temp_i, block_i], ...，把语句 %rx = add i64 temp_i, 0 插入到 block_i 块，然后把这段含有 phi 函数的指令整个删掉，对所有块重复这个过程，就得到去除所有 phi 函数的 LLVM IR（此时又不是 SSA 形式了），可以在此上翻译为 ARM 汇编。

条件分支语句翻译。在我的实现中，生成的 LLVM IR 的比较语句和条件跳转语句一定是一一对应的，而且在对应的比较和条件跳转之间，一定没有其他的比较或者条件跳转语句。这就带来了如下的设计：

```
static LL_instr prevCmpIns = NULL;
void
oldICMP (LL_instr instr, LL_instr instrNxt)
{
  // DBGPRT ("oldICMP, instr: ");
  // LL_print (stderr, instr, Temp_name ());
  Temp_tempList dst = instr->u.OPER.dst;
  Temp_tempList src;
  intarrNumPair_t imms = parseIntImms (instr->u.OPER.assem);
  char *condstr = parseCond (instr->u.OPER.assem);
  if (imms.num == 2)
    /* construct src */
  else if (imms.num == 1)
    /* construct src */
  else
    src = instr->u.OPER.src;
  AS_targets jmps = AS_Targets (instrNxt->u.OPER.jumps->labels);

  emit "cmp `s0, `s1" with src and dst;
  emit "b<cond> `j0" with jmps;
  AS_targets elsejmps
      = AS_Targets (Temp_LabelList (jmps->labels->tail->head, NULL));
  emit "b `j0" with elsejmps;
}
void
ARM_transICMP (LL_instr instr, LL_instr instrNxt)
{
  prevCmpIns = instr;
}
void
ARM_transCJUMP (LL_instr instr, LL_instr instrNxt)
{
  LL_instr prvins = prevCmpIns;
  prevCmpIns = NULL;
  LL_instr insnxt = instr;
  AS_type t = ARM_gettype (prvins);
  if (t == ICMP)
    oldICMP (prvins, insnxt);
  else if (t == FCMP)
    oldFCMP (prvins, insnxt);
}
```

解释：当遇到在前的比较指令，就会调用 ARM_transICMP，给全局变量

`prevCmpIns`（最近一条 LLVM IR 比较指令）赋值当前的 `instr`，此时先不为比较生成对应的 ARM 汇编。因为当前的 LLVM IR 仅仅是把 SSA 中的 phi“推到”上一个块，所以晚些生成比较运算不会影响运行结果。遇到紧接着的条件跳转时，调用 `ARM_transCJUMP`，里面调用 `oldICMP` 生成比较指令以及跳转指令，参数是缓存的前一条比较指令以及当前条件跳转。

另外值得注意的是，汇编语言中需要手动维护函数 frame。函数 prolog 中要依次做这些事情：

1. `push {fp}`，把 caller 的 frame 指针保存到栈上，返回时才能恢复。为了对其 8 位（否则浮点数输出会乱），多 push 一次；
2. `mov fp, sp`，把 frame 指针换成 callee 的 frame 指针，即被调用时刻的栈指针 `sp`；
3. `push {r4-r10}`，`vpush {s16-s31}`，保存所有 callee-saved 寄存器；
4. 把参数移动进合适类型的 `[r/s]0, 1, 2, 3`，我们假定了参数不超过 4 个。

9 活动分析

活动分析主要分为一下三步：

1. 生成控制流图 `fg`。对于普通指令，其后继节点就是下一指令；对跳转指令，其后继节点就是跳转目标或者滑穿（fall through）情况下的下一条指令。依次插入即可。
2. 生成活动图 `ig`，迭代直到每个节点的 live in/out 都没有变化。具体算法参考书上。
3. 生成干涉图 `ig`。活动范围有交叉的节点之间加入一条边即可。需要注意的是，如果使用框架的实现，如前面所述，库中的 degree 结果是错的，需要自己实现度数计算。

10 寄存器分配

本实现中的寄存器分配是比较简易的，只有如下几步：计算节点度数（bg 模块的 degree 函数结果是错的），simplify，spill，分配寄存器，分配栈空间，插入内存空间操作。实质上讲就是只有 simplify，spill 和 color。

用如下结构保存 bg 节点的信息。其实也可以用 bg 节点关联的 `Temp_temp` 作为干涉图节点，但是这样编号会比较大，如果想简便实现（用编号索引数组）就会浪费空间。`G_nodeInfo` 和 `AS_Look_ig` 可以实现相关联的 `Temp_temp` 和 `G_node` 之间互相转化，两者又是一一对应，所以用什么作为索引都可以的。用 `Temp_temp` 作为索引要注意低号码时候的类型，这个时候最好对 int 和 float 分别做一个干涉图。

```
typedef struct REG_igNodeInfo REG_igNodeInfo_t;
struct REG_igNodeInfo
{
  int simpl; /* is simplified. boolean */
```

```
  int *intrf;
  int deg;
  int reg; /* -1: not yet colored, -2: spilt */
  int off; /* -1: not yet assigned */
};
```

要手动计算节点度数是因为，块图的底层实现是有向图，FG_degree 加和了出度和入度，然而块图库并没有保证两个节点之间最多一条边（一个方向），所以库函数算出来的节点度数是错的。以下代码计算了各节点度数。

```
void
initGlobs (G_nodeList ig)
{
  /* ... */
  for (G_nodeList igiter = ig; igiter != NULL;
       igiter = igiter->tail)
    igNodeInfoArr[igiter->head->mykey]
        =  REG_CnstrIgNodeInfo (igiter->head);
  /* ... */
}
```

simplify 过程由函数 REG_rmVertices 进行。每次，该函数调用 REG_rmOneVertex (ig) 移除 ig 节点，直到返回值为 0 说明过程结束。

```
Temp_temp
REG_rmOneVertex (G_nodeList ig)
{
  for (G_nodeList igiter = ig; igiter != NULL; igiter = igiter->tail)
    {
      G_node nd = igiter->head;
      Temp_temp tmp = G_nodeInfo (nd);
      if (igNodeInfoArr[nd->mykey].simpl)
        continue;
      if ((tmp->type == T_int && igNodeInfoArr[nd->mykey].deg < INTK
           && igNodeInfoArr[nd->mykey].reg == -1)
          || (tmp->type == T_float && igNodeInfoArr[nd->mykey].deg < FLOATK
              && igNodeInfoArr[nd->mykey].reg == -1))
        {
          igNodeInfoArr[nd->mykey].simpl = 1;
          decDeg (G_pred (nd), nd);
          decDeg (G_succ (nd), nd);
          return G_nodeInfo (nd);
        }
    }
  return NULL;
}
```

REG_rmOneVertex 找到 ig 中它的迭代第一个遇到的未被移除的节点（simpl 为

真)，如果它的度数小于高限，那么把它的 simpl 标记为真并且相应地减少图度数就完成。同时，每个成功被移除的节点，都压入参数 stack 供后续着色之用。

在 spill 阶段，所有 simpl 标记仍然为假而未被分配寄存器的节点，就是那些在 simplify 迭代之后度数仍然超限之节点，它们的 reg 全部设置为 -2，表示被 spill。

REG_color 对 stack 中每一个节点对应的 temp 分配寄存器。每次着色，都在所有颜色用以下函数找到可用颜色。

```
void
REG_checkRegValid (G_nodeList nl, Temp_temp tmp, int valid[])
{
  for (G_nodeList nliter = nl; nliter != NULL;
       nliter = nliter->tail)
    {
      G_node nd = nliter->head;
      Temp_temp ndtmp = G_nodeInfo (nd);
      if (ndtmp->type == tmp->type
          && igNodeInfoArr[nd->mykey].reg >= 0)
        valid[igNodeInfoArr[nd->mykey].reg] = 0;
    }
}
```

这个函数的 nl 参数会接收 tmp 对应节点的前驱和后续节点列表，然后检查它们中的颜色，选出 valid 中不冲突者。

REG_insertMemop 函数递增地给每个被 spill 的 temp 分配一个栈上相对于 fp 的偏移量，然后在这些 temp 的每次读写操作周围，插入相应的 load 和 store 操作。在此实现中，r8，r9，s29，s30 固定用来 load 源操作数，而 r10，s31 则用来暂时承接目的操作数并 store 到栈上。同时，在函数开头要把栈的大小按照 #spilt temps × arch_size 减下 sp。

比如，冒泡排序的 main 函数开头的开辟栈空间，以及一个被 spill 的 temp 就如此翻译：

regalloc 之前：

```
main:
  ...
  C1:
    movw 280, #0
    mov 212, 280
    movw 281, #32
    mov 213, 281
    ...
```

regalloc 之后：

```
main:
  ...
```

```
C1:
  movw r4, #0
  sub sp, sp, #12
  mov r8, r4
  str r8, [fp, #-100]
  movw r4, #32
  mov r4, r4
  ...
```

caller/callee-saved 寄存器的保存和恢复都在 armgen 时就已完成。对于 callee-saved 寄存器，它们的保存和恢复是在函数 prolog 处 push，返回前 pop。对于 caller-saved 寄存器，`ARM_transCALL` 中是在移入参数之前 push，函数返回、移出 `r0` 之后 pop。以下是冒泡排序的 `main` 调用 `b1$bubbleSort` 前后的行为：

```
...
ldr r5, = b1$bubbleSort
...
push {r0, r1, r2, r3}
vpush {s0, s1, s2, s3, s4, s5, s6, s7}
vpush {s8, s9, s10, s11, s12, s13, s14, s15}
mov r0, r6
ldr r8, [fp, #-104]
mov r1, r8
mov r2, r5
blx r4
vpop {s8, s9, s10, s11, s12, s13, s14, s15}
vpop {s0, s1, s2, s3, s4, s5, s6, s7}
pop {r0, r1, r2, r3}
...
```

11 总结

至此，完成了该 FDMJ 编译器所有步骤的解释。要运行该编译器，按照如下方法：

1. `cd` 到项目根目录；
2. 构建：`make build`；
3. 把需要编译和运行的 FDMJ 源文件（扩展名 `.fmj`）放到 `test` 文件夹中，然后按照目的进行一下操作：
   - 仅编译：`make compile`。注意，该指令默认的 arch size 是 8，如果想要仅编译得到 ARM 汇编文件，可以改 Makefile 中主程序的命令行参数为 4；
   - 编译到并运行 LLVM-IR：`make run-llvm`；
   - 编译到并运行 ARM 汇编：`make run-arm`。