编译实习课程报告

目录

[编译实习课程报告 1](#_Toc11397)

[LAB1 Minijava 1](#_Toc17270)

[1.1 符号表描述 1](#_Toc23095)

[1.2 建立符号表 4](#_Toc27362)

[1.3 类型检查 5](#_Toc17162)

[LAB2 Minijava \_To\_Piglet 6](#_Toc16715)

[2.1 问题分析与设计 6](#_Toc717)

[2.2 翻译方案 7](#_Toc22570)

[LAB3 Piglet\_To\_Spiglet 8](#_Toc24150)

[LAB4 Spiglet\_To\_Kanga 9](#_Toc32057)

[4.1 基本块划分 11](#_Toc32716)

[4.2 活性分析 12](#_Toc16394)

[4.3 寄存器分配 13](#_Toc13917)

[4.4 翻译生成 14](#_Toc16754)

[LAB5 Kangta\_To\_Mips 15](#_Toc23669)

[5.1 栈结构设计 15](#_Toc850)

[5.2 栈结构的维护 16](#_Toc24826)

[5.3 辅助函数设计 17](#_Toc25463)

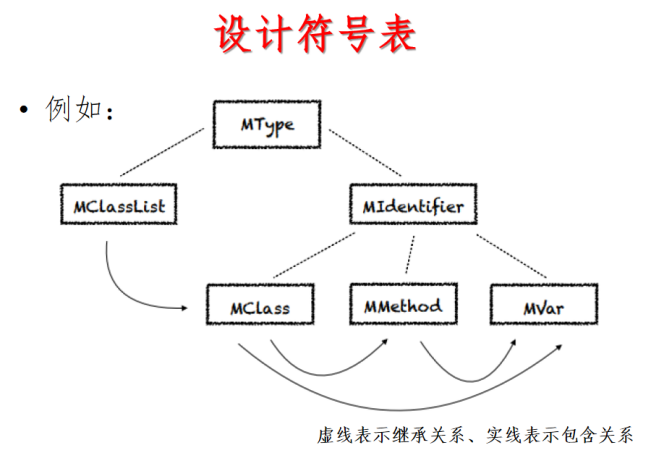
[5.4 一般语句的翻译 18](#_Toc4886)

**LAB1 Minijava**

* 1. **符号表描述**

符号表是一种供编译器用于保存有关源程序构造的各种信息的数据结构。在语法分析过程中，这些信息被逐步地收集并放入到符号表中，然后用于语义检查或者生产目标代码。符号表的每个条目中包含与一个标识符相关的信息，例如关于一个类、一个方法、一个变量等的信息。

在《编译原理》教材中，为支持作用域的嵌套规则，每一个块都设置了符号表，不同层次的符号表通过链接的方法连在一起。但是在本次作业中，minijava的语法规则设定每一个变量声明都在类内或者方法的起始部位，因此不需要考虑方法内部语句块的作用域区分。因此，我使用了老师推荐的简单数据结构来建立符号表框架：



下面分别介绍我对于上面各个数据结构的实现：

|  |  |  |  |
| --- | --- | --- | --- |
| 数据结构 | 描述 | 重要内部变量 | 重要方法 |
| MType | 所有自定义类的基类，方便统一visit函数传递的参数类型 | String kind:可以"int", "boolean", "array", 或者是类名。 | isTypeOf：是否是某类或者其子类；  isSameTypeOf：是否是某类； |
| MClassList | 类的符号表 | HashMap<String,MClass>mclassTable：将每一个类名与一个MClass实例对应。 | insertClass：向符号表中插入一个类；  containVar：根据名字来查找并返回符号表中的类； |
| MIdentifier | 以下三个类的基类 | 无 | 无 |
| MClass | 描述一个类的各种属性信息 | boolean isExtendClass=false:是否是子类  String parentClassName:父类名字  MClass parentClass:父类  HashMap<String,MClass>childClassTable:子类列表  HashMap<String,MMethod>mMethodTable:方法列表  HashMap<String, MVar> mVarTable:变量列表 | insertVar/Method：向相应符号表中插入一个变量/方法；  containVar/containMethod：根据名字来查找并返回相应符号表中的变量/方法；  inheritParentMethod: 继承父类的方法，若发生重载则返回false；  inheritParentVar：继承父类的方法，若发生覆盖则直接跳过（隐藏）；  getVar/MethodIndex：返回变量或者方法在相应符号表中的索引； |
| MMethod | 描述一个方法的各种属性信息 | String methodName:方法名  MClass methodClass:方法所在的类  HashMap<String, MVar> mVarTable:方法内的变量列表  Vector<String> mParameterList:参数列表  Vector<MType> mParameterTypeList:参数类型列表  MType returnType:返回值类型 | insertVar/Parameter：向该方法的变量符号表/变量符号表和参数列表中插入一个变量；  isOverloading：判断是否是某方法的重载；  containVar：根据名字来查找并返回该方法的变量符号表中的变量；  getParamType：返回指定参数的类型，用于判断形参与实参的对应关系；  SetReturnType：设置返回值类型； |
| MVar | 描述一个变量的各种属性信息 | MType varType=null：变量类型  String varName=null：变量名  boolean isClassVar=true：是否是类的字段  MClass varClass=null：类的字段所在的类  MMethod varMethod=null：方法变量所在的方法 | Set/GetVarType：设置/得到变量类型 |

* 1. **建立符号表**

建立符号表的过程是自顶向下构造上述数据结构的过程，也就是在每一个类、方法、变量声明/定义的结点收集相关信息，创建MClass、MMethod、MVar等对象，并放入到对应的符号表中。

具体来说，继承自GJDepthFirst<MType, MType>类，我实现了用于建立符号表的visitor类型：BuildSymbolTableVisitor类，并重载了以下visit函数：

public MType visit(ClassDeclaration n, MType argu)和public MType visit (ClassExtendsDeclaration n, MType argu)用于构造MClass类，并放入到MClasslist的符号表中；

public MType visit(MethodDeclaration n, MType argu)用于构造MMethod类，并放入到(MCLass)argu的符号表中；

public MType visit(VarDeclaration n, MType argu)和public MType visit(FormalParameter n, MType argu)用于构造MVar类，并放入到(MCLass)argu或者的(MMethod)argu符号表中；

public MType visit(MainClass n, MType argu)用于构造main类和main函数的对象；

public MType visit(Type n, MType argu)用于搜集变量的类型信息；

* 1. **类型检查**

根据题目要求，需要完成以下类型检查任务，其检查的时机和算法分别如下：

|  |  |  |
| --- | --- | --- |
| 检查对象 | 检查时机 | 算法 |
| 重复定义类、方法和变量 | 建立符号表过程中 | HashMap的containsKey方法 |
| 类的多重继承 | 建立符号表之后 | minijava的词汇规则保证不会出现 |
| 类的循环继承 | 建立符号表之后 | 1. 更新每个类的子类列表 2. 宽度优先搜索遍历由类和继承关系构成的森林 3. 如果不存在基类（入度为0）或者遍历到之前访问过的结点时，证明存在环 |
| 方法重载 | 检查类的循环继承的同时 | 重载的判断依据是子类有和父类名字相同的方法（已经排除了重复定义的可能）。因此要判断是否存在方法的重载，需要先收集继承关系，因此必须在建立符号表、更新每个类的子类列表之后进行。  确切地说，是在上述的BFS过程中，访问一个节点的叶子节点时，判断子类是否重载了父类的方法。 |
| 使用未定义的类 | 继承检查之后 | 访问Type结点时，判断类名是否在MClasslist中出现；  访问AllocationExpression结点时，判断Identifier子节点是否对应一个未定义的类名。 |
| 使用未定义的变量 | 继承检查之后 | 访问Identifier结点时，判断变量名是否在方法的变量列表，或者方法所在类的变量列表中出现 |
| 使用未定义的方法 | 继承检查之后 | 访问MessageSend结点时，判断被调用的方法是否出现在被调用类的方法列表中 |
| return语句返回类型不匹配 | 访问MethodDeclaration结点时 | 继承自GJDepthFirst<MType, MType>类，我实现了用于类型检查的visitor类型：TypeCheckVisitor类。类型检查的核心思想是，当访问一个表达式时，返回值为它的类型，例如基本类型对应的MClass.kind=”boolean”,”int”,”array”等，类对应于MClassList中的MClass对象。由于minijava严格的词法规则限制，表达式的结构都很简单，因此能够很轻松地得到左值、右值、参数、操作数等表达式的类型，并依照语法规则判断是否不匹配。 |
| 参数不匹配 | 访问MessageSend结点时 |
| 赋值表达式左右操作数类型不匹配 | 访问AssignmentStatement结点时 |
| if、while的判断表达式必须是boolean型 | 访问IfStatement和WhileStatement结点时 |
| Print参数必须为int型 | 访问PrintStatement结点时 |
| 数组下标必须是int型 | 访问ArrayLookup结点时 |
| 操作数相关：+、\*、< 等操作数须为int型 | 访问PlusExpression等结点时 |

**LAB2 Minijava \_To\_Piglet**

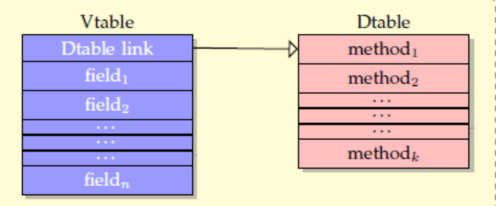
**2.1 问题分析与设计**

这一节要将Minijava源程序翻译成Piglet源程序，关键区别在于Minijava是面向对象的，而Piglet是面向过程的。

因此重点便是为每一个类的对象维护一个方法表和一个属性表，对于字段的读取、方法的调用都通过访问内存中的方法表来实现。

我是按照课件推荐的方法来实现的。具体来说，对于每个对象，其属性表中每4个字节为1个表项，第一个表项存放方法表的地址，后续表项存放对象的属性。方法表中的每一个表项大小为4个字节，存放的是方法名的字符串。

要访问属性或者方法时，只需要通过getIndex获得指定属性或者方法名的索引，然后通过HLOAD等指令就可以读取内容。



**2.2 翻译方案**

为了完成翻译，我以GJDepthFirst<MType, MType>为基类，实现了TranslateToPiglet类，并重载了访问各结点的visit方法。

下面列出翻译过程中的重要步骤和算法。

|  |  |  |
| --- | --- | --- |
| 翻译对象 | 翻译方案 | |
| 函数头 | 函数名改为“类名\_方法名”，参数个数=原来的形参个数+1（参数0是被调用者的属性表地址） | |
| 形参列表 | 不用翻译。但是访问相应结点（formalparameter）时，要完成变量名与TEMP临时变量的绑定（将变量名与”TEMP i”键值对放入到currentFunctionUsedVarList中）。 | |
| 变量声明 | 同上 | |
| 语句/表达式（仅列出部分作为示例） | AllocationExpression | 1. 指定TEMP变量记录即将创建对象的属性表地址和方法表地址 2. 生成形如"MOVE dTableAddr HALLOCATE 8”的指令，为属性表和方法表分配地址 3. 初始化属性表和方法表 4. 返回属性表地址 |
| AssignmentStatement | 分以下两种情况讨论：   1. 左值是类的字段。需要找到该字段在属性表的偏移量和该字段对应的TEMP变量（如果没有则创建），然后不仅要生成对TEMP变量的MOVE指令，还需要生成HSTORE指令，以更新属性表中的值； 2. 左值是方法的变量。只需要找到变量对应的TEMP变量，然后生成相应的MOVE指令。 |
| IfStatement | 添加else和end的label，生成相应的CJUMP指令:  CJUMP expression Label\_else  ...//条件为真时的语句  JUMP Label\_end  Label\_else  ...//条件为假时的语句  Label\_end NOOP |
| MessageSend | 1. 指定TEMP变量用于记录返回值 2. 获得该方法所在对象的属性表和该方法在方法表中的地址 3. 传递实参。以上述属性表地址作为第一个参数，然后生成接下来的不超过18个实参，如果还有其他的实参，那么在内存中分配一块内存，依次存放（HSTORE）在该段内存中，其首地址作为第20个实参。 4. 返回记录了返回值的TEMP变量 |
| ArrayAssignmentStatement | 每一个数组对象，用一个TEMP变量记录其在内存中的首地址（如果是类的字段则还记录在属性表中）。该段内存的第一个字段是数组长度，后面顺序存放数组元素。 |
| 返回值 | 直接翻译，即：”RETURN “字符串加上对expression的翻译。 | |

**LAB3 Piglet\_To\_Spiglet**

spiglet与piglet语言非常接近，代码整体结构基本可以照搬，只需要调整适应一些细节。 具体来说，spiglet语言没有“嵌套表达式”，只有move语句可以使用表达式。因此，翻译方案中重点实现了以下两点：

1. 访问Exp结点时，除非该结点是TEMP实例，否则单独地生成一条MOVE指令.
2. 访问其他含有Exp变量的结点时，首先访问Exp结点，将返回的TEMP地址替换原piglet指令中表达式的位置。

**LAB4 Spiglet\_To\_Kanga**

这一节的任务是将spiglet代码翻译成kanga代码，二者最大的区别是kanga语言的寄存器是有限的，因此最重要的任务就是寄存器分配和栈的使用。

按照老师在课上的提示，我将任务分解成了四个部分：基本块划分、活性分析、寄存器分配、翻译生成。

如此分解的好处是，各个部分相对独立，耦合性低，比较方便调整具体的算法实现。这四个部分所共同使用的数据结构，是对spiglet中的语法成分的描述，包括基类MySpigletType、子类MyGoal、MyProcedure、MyStmt、MyTemp等。下表总结了上述各类的属性和方法。

|  |  |  |  |
| --- | --- | --- | --- |
| 类名 | 描述 | 属性 | 方法 |
| MySpigletType | 基类，作为visit接口传递的参数类型 | 无 | 无 |
| MyGoal | 子类，相当于符号表 | HashMap<String,MyProcedure> table | getProcess |
| MyProcedure | 子类，相当于过程的符号表 | HashMap<Integer, MyTemp> temps:过程出现的temp变量  Vector<MyStmt> stmtlist:语句列表  int paranum,maxparanum:参数个数,调用过程的最多参数  Int spillnum, tempspillnum:溢出单元个数，参数和寄存器分配时溢出的单元个数  Vector<Integer> blockloc: 记录块头指令位置的列表  HashMap<String, Integer> labels: 标签和位置的哈希表 | addstmt  addlabel |
| MyStmt | 子类，记录一条语句的各种属性 | int index: 位置  HashMap<Integer, Integer> livevars: 本句的活跃变量  boolean blockend: 是否块尾  boolean labeled = false: 是否有标签  String label: 标签 | Addlive  Deletelive  Addnextlive  Addjumplive  liveschanged |
| MyTemp | 子类，描述一个temp的寄存器分配情况 | Integer index=0: temp变量的序号  int reg = -1: 寄存器序号  int spill = -1: 溢出单元序号  boolean spilled=false: 是否溢出  HashSet<Integer> interferences: 干扰变量集合 | 无 |

接下来我分别介绍四个部分的实现。

**4.1 基本块划分**

基本块指的是程序流图中的基本结点，也就是只能从块中的第一个指令进入、从最后一个指令离开的连续指令序列。进行基本块划分的目的是方便活性分析，因为如果是基本块内部的指令，则其活性只由其下一条指令和它自身内容决定，基本块的最后一条指令则由基本块的后继结点和它滋镇内容决定。

基本块划分的实现很简单，只需要一趟遍历spiglet的AST即可。对于一个过程，要顺序地访问语句：

1. 如果是第一条指令，则判定为块头
2. 如果是CJUMP或者JUMP语句，则判定为当前块的块尾，下一句为块头。对于目标label，若在已访问label表中，则直接设为块头；否则放入到待访问label表中。
3. 如果当前语句前有Label，若label在待访问label表中，则判定为块头；否则压入到已访问label中
4. 最后一条指令，判定为块尾

表示块头与块尾的数据结构也非常简单，对于每一个过程对象，会有Vector<Integer> blockloc来记录每一个块头指令的索引；对于每一个语句对象，会有boolean blockend来记录是否是块尾。

**4.2 活性分析**

在基本块划分的基础上，进行活性分析。

基本方法是不动点算法，即一直进行活性逆向传递，直到所有语句的活性都不再改变。对应的代码如下：

while(changed) {

changed = false;

for(int i=proc.stmtlen-1;i>=0;--i) {

Node stmt = n.f0.nodes.elementAt(i);

MyStmt mystmt = proc.stmtlist.elementAt(i);

oldlives.clear();

oldlives.putAll(mystmt.livevars);

stmt.accept(this, mystmt);

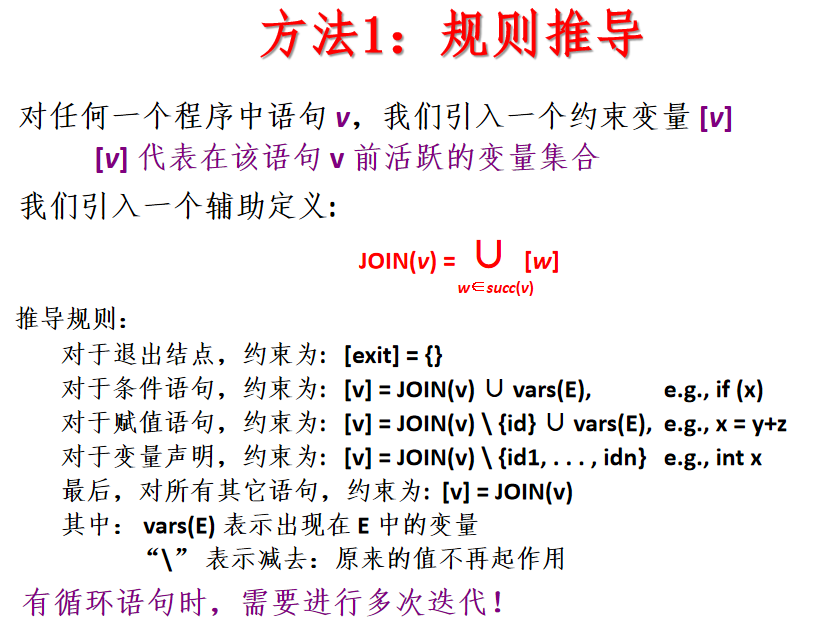
changed = (mystmt.liveschanged(oldlives))?true:changed;

}

}

其中oldlives的作用只是用于和传递之后的活跃变量列表进行比对，以判断是否到达了不动点。

对于具体的语句，我使用了规则推导的方法来计算活性。每一种的计算规则已经在课件中讲的很清楚了（如下图），因此不再赘述。



**4.3 寄存器分配**

关于寄存器分配，老师介绍了两种最常见的方法：基于图着色的方法和基于线性扫描的方法。

我提交的第一个版本使用了线性扫描的方法，但是由于我没有充分理解该算法的含义，造成了冗余。我是在每一个基本块内进行寄存器分配，每次进入和离开块时需要将所有的活跃变量从栈中读入或者写出，所以产生了大量的冗余指令。因此，经过查阅文献，我发现基础的线性扫描算法并不需要考虑活跃变量和基本块，只需要将变量出现的最远区间作为活性区间，然后通过线性扫描的方法进行寄存器分配。但是这样的简单线性扫描live interval方法自然不如基于每个语句的活性分析结果进行寄存器分配算法的效率高，但是更高级的扫描live range的方法我还没有读懂，因此我选择了使用更经典、寄存器利用率更高的图着色算法作为优化。

图着色算法需要构造一个干扰图——以temp变量为结点，以在一个语句同时活跃作为关系——，对于干扰图进行着色。

着色的颜色数量设置为K=18，即包括了s寄存器和t寄存器，而剩下的a寄存器专门用作参数传递，v0寄存器用于返回值传递，v0和v1寄存器用于不活跃的temp变量的使用。

这一部分是在visitor.InterferenceGraph.java中实现，特别是其中的\_initproc方法涵盖了各个子步骤的调用。

1. 首先初始化过程的temps列表。遍历所有的Temp结点，将其和一个MyTemp对应，放入temps索引表（若之前出现过，则跳过）。
2. 构造干扰图。干扰图的数据结构，是temps列表中的每一个MyTemp变量的interference列表，记录了与该temp变量互相干扰的变量。（类似于图的邻接表表示方法）
3. 着色和寄存器分配。使用简单的启发式着色方法：每次删去度数小于K的结点，直到结点数为0或者无法删除（则溢出然后继续）。而实现的数据结构，包括了优先队列（结点关于度数的最小堆，每次删去堆头）、栈（删去的结点放入到栈中，之后着色是从栈顶开始）、列表（记录溢出变量，并分配溢出单元）。
4. 溢出。如上，每次没有度数小于K的结点时，溢出一个结点，然后再次尝试删去，重复操作直到全部都删去或者溢出。对于溢出单元的编号，放在栈中的过程实参（Temp4, Temp5, ...）是编号最小的溢出单元，必须按照参数的顺序来访问；其他的溢出单元就只要有编号就可以。

**4.4 翻译生成**

完成了上述三个部分之后，翻译生成的工作就变得很简单了。

Spiglet和kanga的语法结构有一定相似处，大多数spiglet语句都可以找到对应的一个语句，因而只需要将出现的temp变量替换为mytemp对象记录的寄存器即可。有以下四点需要注意：

1. 溢出的temp变量，需要先增加一个”ALOAD v0/v1 SPILLEDARG I”语句，然后再将temp变量替换为v0/v1.
2. 过程翻译开始前，先将实参加载。如果是前4个实参，则需要从a寄存器”MOVE””到对应的寄存器或者”ASTORE”到对应的溢出单元；如果是之后的实参，本身就放在了溢出单元，只有当有分配的寄存器时，才需要”ALOAD”到对应的寄存器。
3. Kanga的Call单独成一条指令。因此需要先将参数传递到a寄存器或者溢出单元中，然后再生成call指令，之后将原来的”MOVE temp Call ...”翻译成”MOVE reg v0”。
4. kanga没有RETURN语句，因此需要在过程最后添加一个”MOVE v0 ...”或者”ALOAD v0 ...”之类的语句，将返回值专门拷贝到v0寄存器中。

**LAB5 Kangta\_To\_Mips**

**5.1 栈结构设计**

|  |
| --- |
| 参数: f1 arg2(Spilled arg1)  f1 arg1(Spilled arg0) |
| ra | fp指针指向 |
| Old fp |
| 溢出单元: Spilled arg3  Spilled arg2 |
| callee参数: f2 arg2 |
| F2 arg1 | sp指针指向 |

对于每个kanga过程，我设计维护了一个如上图所示的栈结构。fp指针指向栈底，sp指针指向栈顶。而参数由调用者放在调用者的栈顶处。因此，对于一个声明为f1[x][y][z]的kanga过程来说，其栈的长度L由以下几部分组成：

1. ra和old fp。如果是”main”过程，则只有返回值，长度为L1=4；如果是其他过程，则两者都有，长度为L1=8。
2. 部分Spilled arg。对于一个过程，其溢出参数包括了放在栈中的参数数量、保存在栈中的寄存器值等等。由于上述的栈结构设计，所以一个过程的fp与sp之间的spilled arg数量需要减去溢出参数数量。也就是L2=(y-max(0, x-4))\*4.
3. 最大参数数量。如上所述，一个过程的fp与sp之间，还存放了该过程调用的子过程的溢出参数。这部分的长度为L3=max(z-4, 0)\*4

**5.2 栈结构的维护**

在mips语言中，需要显式地维护栈结构，包括修改栈指针的值、通过栈指针访问栈的内容、设置栈的大小等等。下面分别从过程的头部、尾部、参数传递、溢出单元访问等方面进行描述。

当进入一个过程的执行时，首先需要维护栈结构。一般包括以下四个语句：

sw $fp, -8($sp) （main过程不需要这一语句）

move $fp, $sp

subu $sp, $sp, L

sw $ra, -4($fp)

当离开一个过程时，需要恢复栈结构。包括与上面对应的四个语句：  
 lw $ra, -4($fp)

lw $fp, -8($sp) （main过程不需要这一语句）

addu $sp, $sp, L

J $ra

当遇到kanga语言中的pass arg i语法时，需要翻译成对于的栈地址：t($sp)，其中t=4\*(i-1).

当遇到kanga语言中的spilled arg i语法时，需要翻译成对于的栈地址：如果该溢出单元对应于过程的参数，那么地址在上一个过程的栈顶，因此是t($fp)，其中t=4\*i。否则，地址在当前过程栈中的溢出单元部分，因此是t($sp)，其中t=L3+4\*i-L1.

**5.3 辅助函数设计**

针对kanga语言中的hallocate、print、error等指令，我设置了专门的mips过程来进行解决。基本方法是调用syscall指令，根据$a0的不同值调用不同的系统功能，例如内存分配、打印或者中断退出等。当遇到相应的kanga指令时，只需要将其翻译到对指定辅助过程的过程调用即可。

**5.4 一般语句的翻译**

kanga和mips语言在寄存器和指令集合方面有很多相似之处，可以将kanga指令简单地翻译成对应的mips指令。

|  |  |
| --- | --- |
| kanga语句 | mips指令 |
| Noop | nop |
| CJump reg label | beqz reg label |
| Jump label | b label |
| HStore ... | sw ... |
| HLoad ... | lw ... |
| Call ... | jal或者jalr |
| Move ... | move 或者li或者la... |
| ALoad ... | lw ... |
| AStore | sw ... |
| Passarg ... | 见5.2 |
| Spilled arg ... | 见5.2 |
| print ... | 见5.3 |
| error ... | 见5.3 |

**编译实习面测**

错误样例分析:

一、Minijava-piglet

Test15.java 执行错误：空指针

（一）比对java和pg代码，发现错误位置：

Java:

if (i < 0)

k = 0;

else {

System.out.println(k);

...

}

Piglet:

CJUMP LT TEMP 34 0 Label0

HLOAD TEMP 35 TEMP 0 8

MOVE TEMP 35 0

HSTORE TEMP 0 8 TEMP 35

JUMP Label1

Label0

PRINT TEMP 35

（二）分析错误原因：

在生成的pg代码中，在iftrue语句块中，类的字段k与TEMP35绑定了；但是，在else分支中，TEMP35未初始化，所以print会产生空指针报错。

（三）解决办法：

每次遇到类的字段，不管有没有绑定，都生成一个“HLOAD”指令。

在public MType visit(Identifier n, MType argu) 方法中将相关语句修改为：

if (((MVar) id).isClassVar == true) {

MClass varClass = ((MMethod)argu).methodClass;

String temp;

String varOffset = Integer.toString(4 \* (varClass.getVarIndex(name) + 1));

if(currentFunctionUsedVarList.containsKey(name) == false) {

temp = getTemp();

currentFunctionUsedVarList.put(name, temp);

}

else {

temp = currentFunctionUsedVarList.get(name);

}

pigletCode += "\nBEGIN\n";

pigletCode += "HLOAD " + temp + " TEMP 0 " + varOffset + "\n";

pigletCode += "RETURN " + temp + "\n";

pigletCode += "END\n";

}

（之前只考虑了contains==false的情况）

修改后生成的pg代码：

PRINT

BEGIN

HLOAD TEMP 42 TEMP 0 8

RETURN TEMP 42

END

二、Piglet-spiglet

Test20.pg 输出结果错误

Test22.pg

（一）比对pg和spg代码，发现错误位置：

Piglet:

PLUS

BEGIN HLOAD TEMP 100 TEMP 0 8 RETURN TEMP 100 END

BEGIN HLOAD TEMP 100 TEMP 0 12 RETURN TEMP 100 END

Spiglet:

HLOAD TEMP 100 TEMP 0 8

HLOAD TEMP 100 TEMP 0 12

MOVE TEMP 130 PLUS TEMP 100 TEMP 100

（二）分析错误原因：

在piglet代码中，不同的begin-return-end语句可以返回同一个temp，对应不同的值，而spiglet不同。

（三）解决办法：

添加一个move语句，将return对象返回到一个新的temp上，避免重复。

/\* f0 -> "BEGIN" f1 -> StmtList() f2 -> "RETURN" f3 -> Exp() f4 -> "END"\*/

public String visit(StmtExp n) {

n.f1.accept(this);

String exp = n.f3.accept(this);

String temp = getTemp();//add

spigletCode += "MOVE " + temp + " " + exp + "\n";//add

return temp;//return exp;

}

三、进阶测试分析

（一）没有实现短路与

解决办法：应该把简单的一行“LT ”代码，改为先判断左边的值是否为0，如果为0，则直接进入到else语句中；不然再判断右边的值是否0.

（二）加载溢出的参数到寄存器时，分配代码出现了bug。

有一行判断v1寄存器是否使用过的代码被误删了，在特殊情况下运行到该段代码会出现计算错误（将不同的溢出变量加载到了同一个v0寄存器中，导致了v1的值计算错误，进而导致下面在计算数组下标时计算错误，出现了访问栈溢出现象）。

解决办法：恢复被误删的代码即可。