扩展：可变变量

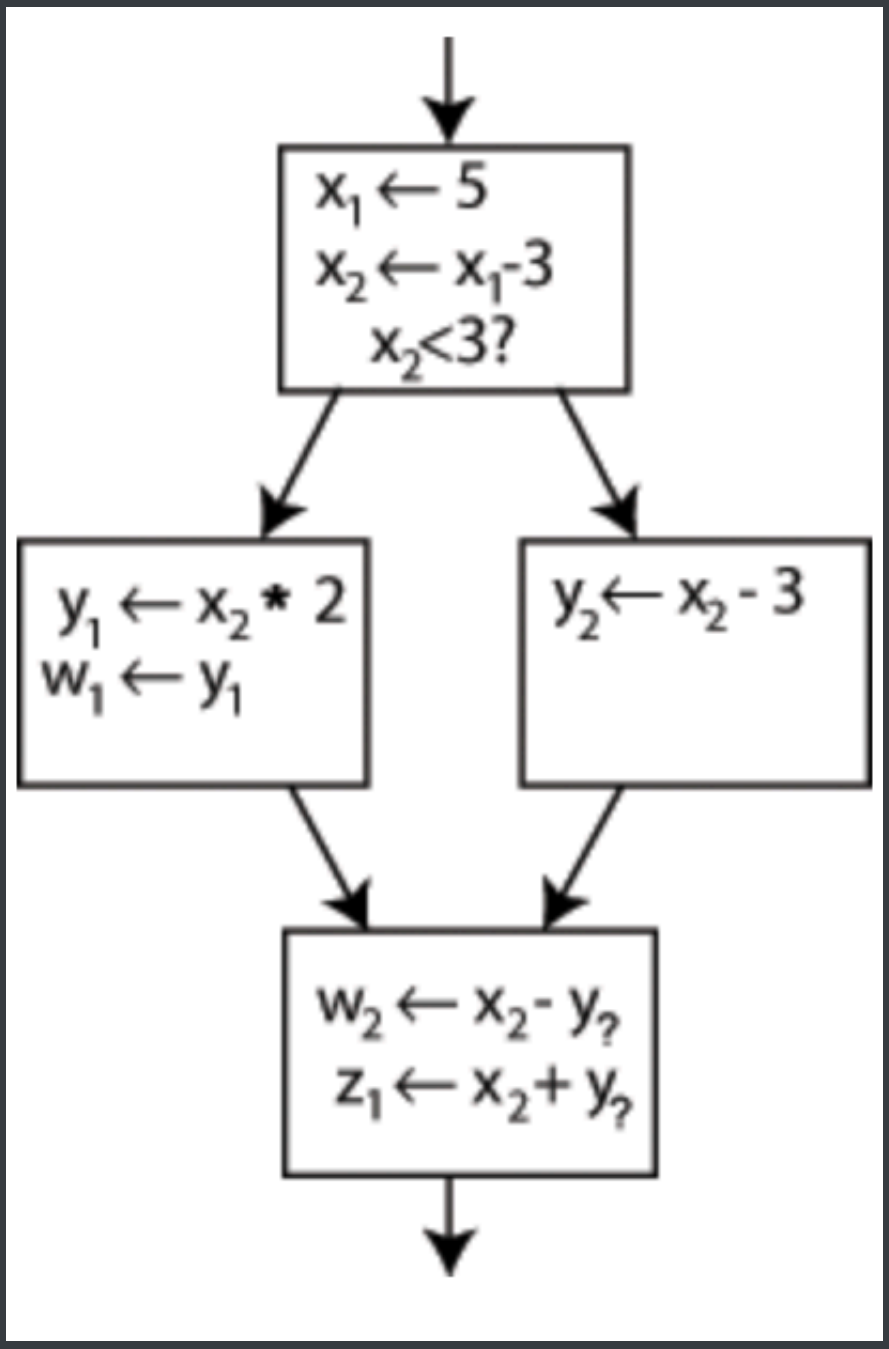
by 王子昂 蒋颖

一、背景知识

SSA

静态单赋值，Static Single-Assignment，这是一种中间表示形式。 之所以称之为单赋值，是因为每个名字在SSA中仅被赋值一次。为了得到 SSA 形式的 IR，起初的 IR 中的变量会被分割成不同的版本（version），每个定义（definition：静态分析术语，可以理解为赋值）对应着一个版本。在教科书中，通常会在旧的变量名后加上下标构成新的变量名，这也就是各个版本的名字。显然，在 SSA 形式中，UD 链（Use-Define Chain）是十分明确的。也就是说，变量的每一个使用（use：静态分析术语，可以理解为变量的读取）点只有唯一一个定义可以到达。

```
//例子
 y := 1
 y := 2
 x := y
 //对于采用非 SSA 形式 IR 的编译器来说，它需要做数据流分析（具体来说是到达-定义分析）来确定选取哪一行的 y 值。但是对于 SSA 形式来说，不需要做数据流分析就可以知道第三行中使用的 y 来自于第二行的定义。
 y1 := 1
 y2 := 2
 x1 := y2
```



上图中第四个基本块的两个个前驱基本块里都对 y 进行了定义，这里我们并不知道程序最终会从哪个前驱基本块到达该基本块。那么，我们如何知道 y 该取哪个版本?

构造静态单赋值形式的过程会在CFG中的每个汇合点之后插入phi函数，汇合点即为CFG中多条代码路径汇合之处。在汇合点处，不同的静态单赋值形式名必须调和为一个名字。整个过程大致为两步：

(1) 插入PHI函数

在具有多个前趋的每个程序块起始处，插入相应的PHI函数。不同风格的静态单赋值的形式，插入PHI函数的条件不同。

最小静态单赋值形式：在任何汇合点处插入一个PHI函数，只要对应于同一原始名字的两个不同定义汇合，就插入符合静态单赋值形式定义、数目最少的PHI函数。

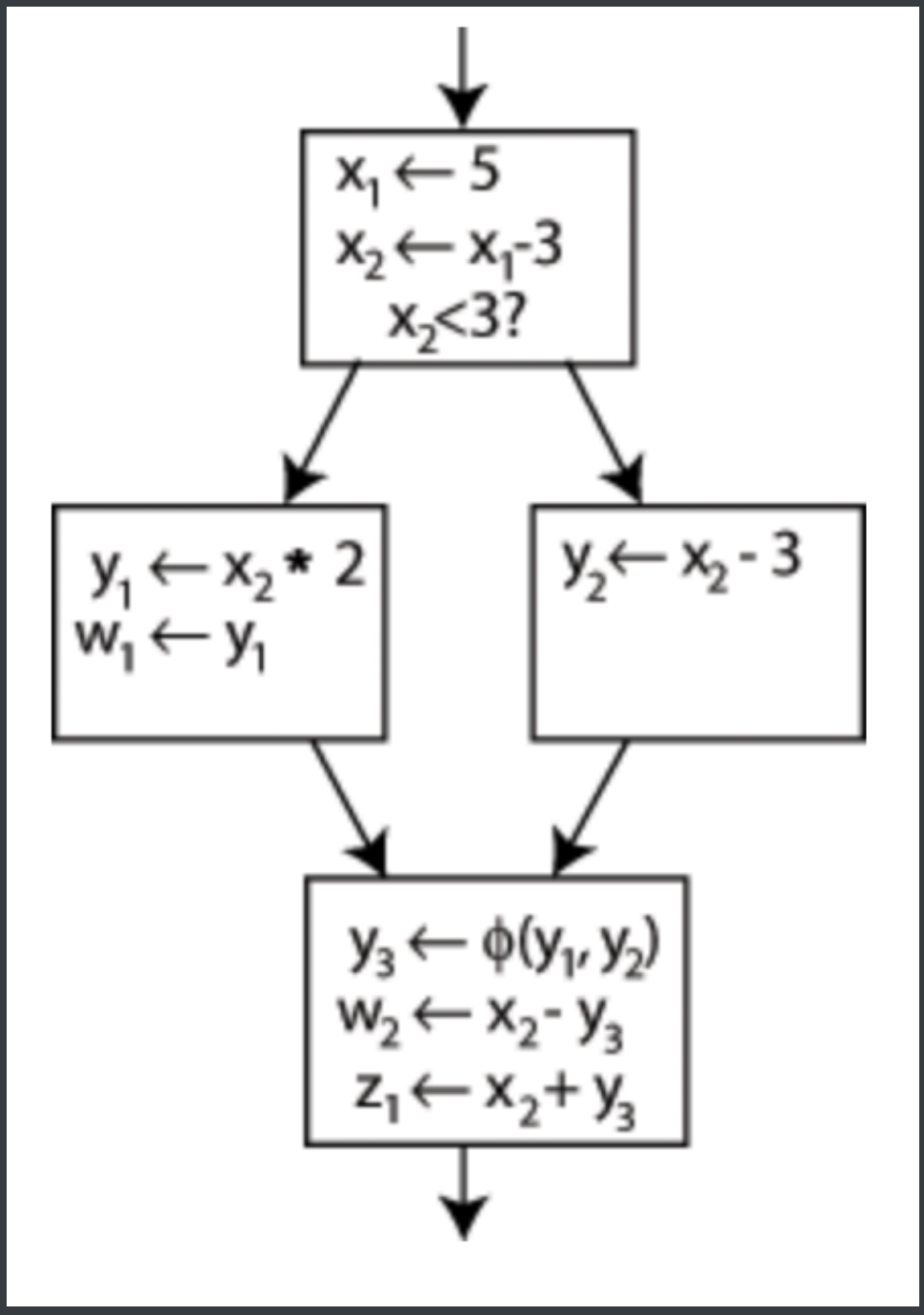
最大静态单赋值形式：在每个汇合点处为每个变量放置一个PHI函数。

剪枝静态单赋值形式：向PHI函数插入算法中添加一个活跃性判断，以避免添加死亡的PHI函数。构造过程必须计算LiveOut集合，因此构建静态单赋值形式的代价高于构建最小静态单赋值形式。

半剪枝静态单赋值形式：这是最小静态单赋值形式和剪枝静态单赋值形式之间的一种折中。在插入PHI函数之前，算法先删除那些只在def的基本块中活跃的变量，因为只在一个基本块中活跃的变量，是不用插入与之对应的PHI函数的

(2) 重命名

在插入PHI函数之后，编译器可以计算可达定义。由于插入的PHI函数也是定义，它们确保了对任一使用处都只有一个定义能够到达。接下来，编译器可以重命名每个使用处的变量和临时值，以反映到达该处的定义。



示例

```
//c代码
int G, H;
int test(_Bool Condition) {
  int X;
  if (Condition)
    X = G;
  else
    X = H;
  return X;
}
//对应的LLVM IR
@G = weak global i32 0   ; //type of @G is i32*
@H = weak global i32 0   ; //type of @H is i32*

define i32 @test(i1 %Condition) {
entry:
  br i1 %Condition, label %cond_true, label %cond_false

cond_true:
  %X.0 = load i32* @G
  br label %cond_next

cond_false:
  %X.1 = load i32* @H
  br label %cond_next

cond_next:
  %X.2 = phi i32 [ %X.1, %cond_false ], [ %X.0, %cond_true ]
  ret i32 %X.2
}
//为了合并传入的值，cond_next块中的X.2 phi节点根据控制流来自何处来选择要使用的正确值：如果控制流来自cond_false块，则X.2获取X的值0.1。或者，如果控制流来自cond_true，则它获得X.0的值。
```

二、LLVM中的内存

虽然LLVM确实要求所有寄存器值都是SSA形式，但它不允许存储器对象采用SSA形式。在上面的示例中，请注意G和H的负载是对G和H的直接访问：它们不会重命名或版本化。

为每个可变对象创建一个堆栈变量（它存在于内存中，因为它在堆栈中），LLVM的堆栈变量由LLVM alloca指令声明。

使用alloca指令分配的堆栈内存是完全通用的：可以将堆栈槽的地址传递给函数，可以将其存储在其他变量中。

示例重写

```
@G = weak global i32 0   ; type of @G is i32*
@H = weak global i32 0   ; type of @H is i32*

define i32 @test(i1 %Condition) {
entry:
  %X = alloca i32           ; //type of %X is i32*.
  br i1 %Condition, label %cond_true, label %cond_false

cond_true:
  %X.0 = load i32* @G
  store i32 %X.0, i32* %X   ; //Update X
  br label %cond_next

cond_false:
  %X.1 = load i32* @H
  store i32 %X.1, i32* %X   ; //Update X
  br label %cond_next

cond_next:
  %X.2 = load i32* %X       ; //Read X
  ret i32 %X.2
}
```

结论

处理任意可变变量无需创建Phi节点的方法：

1.每个可变变量都成为堆栈分配。 2.每次读取变量都成为堆栈的负载。 3.变量的每次更新都成为堆栈的存储。 4.获取变量的地址只是直接使用堆栈地址。

问题

引入了大量的堆栈流量，影响性能

解决

mem2reg优化，将这样的内存分配提升到SSA寄存器中，并根据需要插入Phi节点。

通过传递运行此示例：

```
$ llvm-as < example.ll | opt -mem2reg | llvm-dis
@G = weak global i32 0
@H = weak global i32 0

define i32 @test(i1 %Condition) {
entry:
  br i1 %Condition, label %cond_true, label %cond_false

cond_true:
  %X.0 = load i32* @G
  br label %cond_next

cond_false:
  %X.1 = load i32* @H
  br label %cond_next

cond_next:
  %X.01 = phi i32 [ %X.1, %cond_false ], [ %X.0, %cond_true ]
  ret i32 %X.01
}
```

mem2reg仅适用于某些情况下的变量：

1. mem2reg是alloca驱动的：它查找alloca，如果它可以处理它们，它会改进它们。它不适用于全局变量或堆分配。
2. mem2reg仅在函数的入口块中查找alloca指令。在入口块中保证alloca只执行一次，这使得分析更简单。
3. mem2reg仅改进直接加载和存储的alloca。如果将堆栈对象的地址传递给函数，或者涉及任何指针算法，则不会改进alloca。
4. mem2reg仅适用于第一类值的分配（例如指针，标量和向量），并且仅当分配的数组大小为1时才有效。 mem2reg无法将结构或数组提升为寄存器。“sroa”传递更强大，并且在许多情况下可以提升结构，“联合”和数组。

三、VSL中的可变变量

（1）符号表在代码生成时由“NamedValues”映射进行管理，此时需要“NamedValues”保存可变变量的内存位置。

```
extern std::map<std::string, AllocaInst*> NamedValues;
```

（2）辅助函数：确保在函数的入口块中创建alloca

```
/// CreateEntryBlockAlloca - Create an alloca instruction in the entry
block of
/// the function.  This is used for mutable variables etc.
static AllocaInst *CreateEntryBlockAlloca(Function *TheFunction,
                                          const std::string &VarName)
{
  IRBuilder<> TmpB(&TheFunction->getEntryBlock(),
                 TheFunction->getEntryBlock().begin());
  return TmpB.CreateAlloca(Type::getDoubleTy(TheContext), 0,
                           VarName.c_str());
}
//创建了一个IRBuilder对象，它指向入口块的第一条指令（.begin（））。然后它创建一个
具有预期名称的alloca并返回它。
```

（3）变量存在于堆栈中，生成对它们的引用的代码需要从堆栈槽生成负载：

```
Value *VariableExprAST::codegen() {
  // Look this variable up in the function.
  Value *V = NamedValues[Name];
  if (!V)
    return LogErrorV("Unknown variable name");

  // Load the value.
  return Builder.CreateLoad(V, Name.c_str());
}
```

（4）更新定义变量的内容以设置alloca，以WhileStatAST::codegen()为例：

```
Function *TheFunction = Builder.GetInsertBlock()->getParent();

// Create an alloca for the variable in the entry block.
AllocaInst *Alloca = CreateEntryBlockAlloca(TheFunction, VarName);

// Emit the start code first, without 'variable' in scope.
Value *StartVal = WhileCondition->codegen();
    if (!StartVal)
        return nullptr;

StartVal = Builder.CreateFCmpONE(StartVal, ConstantFP::get(TheContext,
APFloat(0.0)), "whilecond");

// Store the value into the alloca.
```

```
Builder.CreateStore(StartVal, Alloca);
...

// Compute the end condition.
Value *EndCond = WhileCondition->codegen();
    if (!EndCond)
        return nullptr;
EndCond = Builder.CreateFCmpONE(EndCond, ConstantFP::get(TheContext,
APFloat(0.0)), "whilecond");

// Reload, increment, and restore the alloca.  This handles the case
where the body of the loop mutates the variable.

Value *CurVar = Builder.CreateLoad(Alloca);
Value *NextVar = Builder.CreateFAdd(CurVar, StepVal, "nextvar");
Builder.CreateStore(NextVar, Alloca);
...
```

与之前的区别：我们不再需要构建一个PHI节点，我们使用load / store来根据需要访问变量。

（5）给可变参数变量分配内存：

```
Function *FunctionAST::codegen() {
  ...
  Builder.SetInsertPoint(BB);

  // Record the function arguments in the NamedValues map.
  NamedValues.clear();
  for (auto &Arg : TheFunction->args()) {
    // Create an alloca for this variable.
    AllocaInst *Alloca = CreateEntryBlockAlloca(TheFunction,
Arg.getName());

    // Store the initial value into the alloca.
    Builder.CreateStore(&Arg, Alloca);

    // Add arguments to variable symbol table.
    NamedValues[Arg.getName()] = Alloca;
  }

  if (Value *RetVal = Body->codegen()) {
    ...
```

（6）添加mem2reg传递

```
// Promote allocas to registers.
TheFPM->add(createPromoteMemoryToRegisterPass());
// Do simple "peephole" optimizations and bit-twiddling optzns.
TheFPM->add(createInstructionCombiningPass());
// Reassociate expressions.
TheFPM->add(createReassociatePass());
...
```

四、新增的赋值操作符

我们像处理其它二元操作符一样，来处理赋值操作符 `=`

设置优先级

```
int main() {
  // 1 是最小的优先级
  BinopPrecedence['='] = 2;
  BinopPrecedence['<'] = 10;
  BinopPrecedence['+'] = 20;
  BinopPrecedence['-'] = 20;
```

中间代码生成

```
Value *BinaryExprAST::codegen() {
  // '=' 情况：作为一种特例处理，因为在赋值情况下我们不将LHS当作表达式
  if (Op == '=') {
    // 赋值操作中我们将LHS当作标识符
    // 假定不使用运行时类型识别（RTTI）这是LLVM的默认生成方式
    // =如果希望进行运行时类型识别，可以使用 dynamic_cast 来进行动态错误检查
    VariableExprAST *LHSE = static_cast<VariableExprAST *>(LHS.get());
    if (!LHSE)
      return LogErrorV("destination of '=' must be a variable");
    // 生成 RHS 部分代码
    Value *Val = RHS->codegen();
    if (!Val)
      return nullptr;

    // 寻找变量名
    Value *Variable = NamedValues[LHSE->getName()];
    if (!Variable)
```

```
        return LogErrorV("Unknown variable name");

      Builder.CreateStore(Val, Variable);
      return Val;
    }
```

五、用户定义的局部变量

扩展词法分析

```
enum Token {
    ...
    VAR = -16,
    ...
};

int recKeyword() {
    ...
    else if (IdentifierStr == "VAR")
        return VAR;
}
```

定义AST节点

```
/// VarExprAST - Expression class for VAR
class VarExprAST : public ExprAST {
  // 保存定义的变量名，以及初始值
  std::vector<std::pair<std::string, std::unique_ptr<ExprAST>>>
VarNames;
  std::unique_ptr<StatAST> Body;

public:
  VarExprAST(std::vector<std::pair<std::string,
std::unique_ptr<ExprAST>>> VarNames,
             std::unique_ptr<StatAST> Body)
    : VarNames(std::move(VarNames)), Body(std::move(Body)) {}

  Value *codegen() override;
};
```

- 可以一次定义多个变量，变量可以拥有初始值（可选）
- `Body` 中允许访问 `VAR` 定义的变量

扩展语法分析

```
std::unique_ptr<ExprAST> ParsePrimary() {
    switch (CurTok) {
    default:
        return LogError("unknown token when expecting an expression");
    case VARIABLE:
        return ParseIdentifierExpr();
    case INTEGER:
        return ParseNumberExpr();
    case VAR:
        return ParseVarExpr();
    case '(':
        return ParseParenExpr();
    case '-':
        return ParseMinusExpr();
    }
}
```

定义 `ParseVarExpr` 方法：

```
/// varexpr ::= 'var' identifier ('=' expression)
static std::unique_ptr<ExprAST> ParseVarExpr() {
    getNextToken(); // eat the var.
    /**
    * 变量声明、初始化部分代码
    */
    std::vector<std::pair<std::string, std::unique_ptr<ExprAST>>>
VarNames;

    // 至少需要一个变量名
    if (CurTok != VARIABLE)
        return LogError("expected identifier after var");

    while (true) {
        std::string Name = IdentifierStr;
        getNextToken(); // eat identifier.

        // 读取可能存在的初始化表达式
        std::unique_ptr<ExprAST> Init = nullptr;
        if (CurTok == '=') {
            getNextToken(); // eat the '='.
```

```
            Init = ParseExpression();
            if (!Init)
                return nullptr;
        }

        VarNames.push_back(std::make_pair(Name, std::move(Init)));

        // 声明变量部分结束，退出循环
        if (CurTok != ',')
            break;
        getNextToken(); // eat the ','.

        if (CurTok != VARIABLE)
            return LogError("expected identifier list after var");
    }


    /**
    * 处理Body部分代码
    */
    auto Body = ParseExpression();
    if (!Body)
        return nullptr;

    return llvm::make_unique<VarExprAST>(std::move(VarNames),
std::move(Body));
}
```

扩展中间代码生成

```
Value *VarExprAST::codegen() {
  std::vector<AllocaInst *> OldBindings;

  Function *TheFunction = Builder.GetInsertBlock()->getParent();

  // 注册所有的变量并进行初始化
  for (unsigned i = 0, e = VarNames.size(); i != e; ++i) {
    const std::string &VarName = VarNames[i].first;
    ExprAST *Init = VarNames[i].second.get();

    // 在将变量添加到作用于前获得初始化表达式，防止初始化表达式中使用变量本身
    Value *InitVal;
    if (Init) {
      InitVal = Init->codegen();
```

```
      if (!InitVal)
        return nullptr;
    } else { // 如果没有指定，赋值为 0.0.
      InitVal = ConstantFP::get(TheContext, APFloat(0.0));
    }
    // 创建 alloca
    AllocaInst *Alloca = CreateEntryBlockAlloca(TheFunction, VarName);
    Builder.CreateStore(InitVal, Alloca);

    // 将该变量的先前值存入OldBindings中，以便在该作用域结束后恢复
    OldBindings.push_back(NamedValues[VarName]);

    // 记录此次绑定的值
    NamedValues[VarName] = Alloca;
  }

  // 生成body部分的代码，现在所有定义的变量均在作用域中
  Value *BodyVal = Body->codegen();
  if (!BodyVal)
    return nullptr;

  // 删除当前作用域中的所有的变量
  for (unsigned i = 0, e = VarNames.size(); i != e; ++i)
    // 恢复原来的值
    NamedValues[VarNames[i].first] = OldBindings[i];

  // 返回Body部分的计算结果
  return BodyVal;
}
```