**第七章阅读笔记**

**---扩展语言:可变变量**

1. **理解可变变量为什么会导致SSA构造的复杂性.**

考虑下列C示例:

Int G, H;int test(\_Bool Condition) {

Int X;

**If** (Condition)

X = G;

**else**

X = H;

**return** X;}

在这种情况下，我们有变量“X”，其值取决于程序中执行的路径。因为在返回指令之前X有两个不同的可能值，所以插入PHI节点以合并这两个值。我们在这个例子中想要的LLVM IR如下所示：

@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}

在此示例中，来自G和H全局变量的加载在LLVM IR中是显式的，并且它们位于if语句的then / else分支中（cond\_true / cond\_false）。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.这里的问题是LLVM 要求其IR处于SSA形式：它没有“非ssa”模式。然而，SSA构造需要非平凡的算法和数据结构，因此每个前端必须重现这种逻辑是不方便和浪费的。

1. **LLVM中的内存**

这里的“技巧”是，虽然LLVM确实要求所有寄存器值都是SSA形式，但它不要求（或允许）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。这与其他一些尝试版本内存对象的编译器系统不同。在LLVM中，不是将内存的数据流分析编码到LLVM IR中，而是使用按需计算的[Analysis Passes](http://llvm.org/docs/WritingAnLLVMPass.html)进行处理。

考虑到这一点，高级想法是我们想要为函数中的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）。为了利用这个技巧，我们需要讨论LLVM如何表示堆栈变量。

在LLVM中，所有内存访问都是使用加载/存储指令显式的，并且经过精心设计，不具备（或需要）“address-of”运算符。注意@ G / @ H全局变量的类型实际上是“i32 \*”，即使变量定义为“i32”。这意味着@G 在全局数据区域中为i32 定义了空间，但其 名称实际上是指该空间的地址。堆栈变量以相同的方式工作，除了使用[LLVM alloca指令](http://llvm.org/docs/LangRef.html" \l "alloca-instruction)声明它们而不是使用全局变量定义声明它们之外：

**define i32** @example(){**entry:**

%X = **alloca i32** *; type of %X is i32\*.*

...

%tmp = **load i32**\* %X *; load the stack value %X from the stack.*

%tmp2 = **add i32** %tmp,1*; increment it*

**Store i32** %tmp2, **i32**\* %X*; store it back*

...

此代码显示了如何在LLVM IR中声明和操作堆栈变量的示例。使用alloca指令分配的堆栈内存是完全通用的：您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等。在上面的示例中，我们可以重写示例以使用alloca技术来避免使用PHI节点：

@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}

有了这个，我们发现了一种处理任意可变变量的方法，而不需要创建Phi节点：

1. 每个可变变量都成为堆栈分配。
2. 每次读取变量都会成为堆栈的负载。
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。

虽然这个解决方案解决了我们当前的问题，但它引入了另一个问题：我们现在显然已经为非常简单和常见的操作引入了大量的堆栈流量，这是一个主要的性能问题。对我们来说幸运的是，LLVM优化器有一个名为“mem2reg”的高度优化的优化过程，可以处理这种情况，将这样的分配提升到SSA寄存器中，并根据需要插入Phi节点。例如，如果您通过传递运行此示例，您将获得：

$ 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传递实现了用于构造SSA形式的标准“迭代优势边界”算法，并且具有许多加速（非常常见）简并情况的优化。mem2reg优化传递是处理可变变量的答案，我们强烈建议您依赖它。请注意，mem2reg仅适用于某些情况下的变量：

1. mem2reg是alloca驱动的：它查找allocas，如果它可以处理它们，它会提升它们。它不适用于全局变量或堆分配。
2. mem2reg只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只提升其使用是直接加载和存储的allocas。如果将堆栈对象的地址传递给函数，或者涉及任何有趣的指针算法，则不会提升alloca。
4. mem2reg仅适用于[第一类](http://llvm.org/docs/LangRef.html" \l "first-class-types)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（或.ll文件中缺失）时才有效。mem2reg无法将结构或数组提升为寄存器。请注意，“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。

对于大多数命令式语言来说，所有这些属性都很容易满足，我们将在下面用Kaleidoscope进行说明。你可能会问的最后一个问题是：我应该为我的前端烦恼吗？如果我直接进行SSA构建，避免使用mem2reg优化传递会不会更好？简而言之，我们强烈建议您使用此技术来构建SSA表单，除非有非常好的理由不这样做。使用这种技术是：

* 经验证且经过充分测试：clang将此技术用于本地可变变量。因此，LLVM最常见的客户端使用它来处理大量变量。您可以确保快速找到错误并尽早修复。
* 速度极快：mem2reg有许多特殊情况，可以在常见情况下快速完成。例如，它具有仅用于单个块的变量的快速路径，仅具有一个分配点的变量，避免插入不需要的phi节点的良好启发式等。
* 调试信息生成需要：[LLVM中的调试信息](http://llvm.org/docs/SourceLevelDebugging.html)依赖于公开变量的地址，以便可以将调试信息附加到它。这种技术与这种调试信息风格非常吻合。

如果不出意外，这样可以更轻松地启动和运行前端，并且实现起来非常简单。

3.这里的“技巧”是，虽然LLVM确实要求所有寄存器值都是SSA形式，但它不要求（或允许）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。这与其他一些尝试版本内存对象的编译器系统不同。在LLVM中，不是将内存的数据流分析编码到LLVM IR中，而是使用按需计算的[Analysis Passes](http://llvm.org/docs/WritingAnLLVMPass.html)进行处理。

考虑到这一点，高级想法是我们想要为函数中的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）。为了利用这个技巧，我们需要讨论LLVM如何表示堆栈变量。

在LLVM中，所有内存访问都是使用加载/存储指令显式的，并且经过精心设计，不具备（或需要）“address-of”运算符。注意@ G / @ H全局变量的类型实际上是“i32 \*”，即使变量定义为“i32”。这意味着@G 在全局数据区域中为i32 定义了空间，但其 名称实际上是指该空间的地址。堆栈变量以相同的方式工作，除了使用[LLVM alloca指令](http://llvm.org/docs/LangRef.html" \l "alloca-instruction)声明它们而不是使用全局变量定义声明它们之外：

**Define i32** @example() {**entry:**

%X = **alloca i32** *; type of %X is i32\*.*

...

%tmp = **load i32**\* %X*; load the stack value %X from the stack.*

%tmp2 = **add i32** %tmp, 1*; increment it*

**Store i32** %tmp2, **i32**\* %X *; store it back*

...

此代码显示了如何在LLVM IR中声明和操作堆栈变量的示例。使用alloca指令分配的堆栈内存是完全通用的：您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等。在上面的示例中，我们可以重写示例以使用alloca技术来避免使用PHI节点：

@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}

有了这个，我们发现了一种处理任意可变变量的方法，而不需要创建Phi节点：

1. 每个可变变量都成为堆栈分配。
2. 每次读取变量都会成为堆栈的负载。
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。

虽然这个解决方案解决了我们当前的问题，但它引入了另一个问题：我们现在显然已经为非常简单和常见的操作引入了大量的堆栈流量，这是一个主要的性能问题。对我们来说幸运的是，LLVM优化器有一个名为“mem2reg”的高度优化的优化过程，可以处理这种情况，将这样的分配提升到SSA寄存器中，并根据需要插入Phi节点。例如，如果您通过传递运行此示例，您将获得：

$ 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传递实现了用于构造SSA形式的标准“迭代优势边界”算法，并且具有许多加速（非常常见）简并情况的优化。mem2reg优化传递是处理可变变量的答案，我们强烈建议您依赖它。请注意，mem2reg仅适用于某些情况下的变量：

1. mem2reg是alloca驱动的：它查找allocas，如果它可以处理它们，它会提升它们。它不适用于全局变量或堆分配。
2. mem2reg只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只提升其使用是直接加载和存储的allocas。如果将堆栈对象的地址传递给函数，或者涉及任何有趣的指针算法，则不会提升alloca。
4. mem2reg仅适用于[第一类](http://llvm.org/docs/LangRef.html" \l "first-class-types)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（或.ll文件中缺失）时才有效。mem2reg无法将结构或数组提升为寄存器。请注意，“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。

对于大多数命令式语言来说，所有这些属性都很容易满足，我们将在下面用Kaleidoscope进行说明。你可能会问的最后一个问题是：我应该为我的前端烦恼吗？如果我直接进行SSA构建，避免使用mem2reg优化传递会不会更好？简而言之，我们强烈建议您使用此技术来构建SSA表单，除非有非常好的理由不这样做。使用这种技术是：

* 经验证且经过充分测试：clang将此技术用于本地可变变量。因此，LLVM最常见的客户端使用它来处理大量变量。您可以确保快速找到错误并尽早修复。
* 速度极快：mem2reg有许多特殊情况，可以在常见情况下快速完成。例如，它具有仅用于单个块的变量的快速路径，仅具有一个分配点的变量，避免插入不需要的phi节点的良好启发式等。
* 调试信息生成需要：[LLVM中的调试信息](http://llvm.org/docs/SourceLevelDebugging.html)依赖于公开变量的地址，以便可以将调试信息附加到它。这种技术与这种调试信息风格非常吻合。

如果不出意外，这样可以更轻松地启动和运行前端，并且实现起来非常简单。

1. **[Kaleidoscope](http://llvm.org/docs/tutorial/LangImpl07.html" \l "mutable-variables-in-kaleidoscope)中的可变变量**

我们将添加两个功能：

1. 使用'='运算符变换变量的能力。
2. 定义新变量的能力。

虽然第一项实际上是关于这一点的，但我们只有传入参数和归纳变量的变量，并且重新定义那些只是到目前为止:)。此外，无论是否要改变它们，定义新变量的能力都是有用的。这是一个激励性的例子，展示了我们如何使用这些：

*# Define ':' for sequencing: as a low-precedence operator that ignores operands# and just returns the RHS.***def** binary : 1 (x y) y;

*# Recursive fib, we could do this before.***def** fib(x)

**If** (x < 3) then

1

**else**

fib(x-1)+fib(x-2);

*# Iterative fib.***def** fibi(x)

Var a = 1, b = 1, c **in**

(**for** i = 3, i < x **in**

c = a + b :

a = b :

b = c) :

b;

*# Call it.*fibi(10);

为了改变变量，我们必须改变现有的变量来使用“alloca技巧”。一旦我们有了这个，我们将添加我们的新运算符，然后扩展Kaleidoscope以支持新的变量定义。

**4.新的赋值运算符**

使用我们当前的框架，添加新的赋值运算符非常简单。我们将像任何其他二元运算符一样解析它，但在内部处理它（而不是允许用户定义它）。第一步是设置优先级：

Int main() {

*// Install standard binary operators.*

*// 1 is lowest precedence.*

BinopPrecedence['='] = 2;

BinopPrecedence['<'] = 10;

BinopPrecedence['+'] = 20;

BinopPrecedence['-'] = 20;

现在解析器知道二元运算符的优先级，它负责所有解析和AST生成。我们只需要为赋值运算符实现codegen。这看起来像：

Value \*BinaryExprAST::codegen() {

*// Special case '=' because we don't want to emit the LHS as an expression.*

**If** (Op == '=') {

*// Assignment requires the LHS to be an identifier.*

VariableExprAST \*LHSE = **dynamic\_cast**<VariableExprAST\*>(LHS.get());

**If** (!LHSE)

**Return** LogErrorV("destination of '=' must be a variable");

与其余的二元运算符不同，我们的赋值运算符不遵循“发出LHS，发出RHS，执行计算”模型。因此，在处理其他二元运算符之前，它将作为特殊情况处理。另一个奇怪的是它需要LHS作为变量。“（x + 1）= expr”无效 - 只允许使用“x = expr”之类的东西。

*// Codegen the RHS.*

Value \*Val = RHS->codegen();

**If** (!Val)

**Return nullptr**;

*// Look up the name.*

Value \*Variable = NamedValues[LHSE->getName()];

**If** (!Variable)

**Return** LogErrorV("Unknown variable name");

Builder.CreateStore(Val, Variable);

**return** Val;}...

一旦我们得到了变量，codegen'的赋值很简单：我们发出赋值的RHS，创建一个商店，然后返回计算值。返回值允许链接赋值，例如“X =（Y = Z）”。

现在我们有了一个赋值运算符，我们可以改变循环变量和参数。例如，我们现在可以运行如下代码：

*# Function to print a double.*extern printd(x);

*# Define ':' for sequencing: as a low-precedence operator that ignores operands# and just returns the RHS.***def** binary : 1 (x y) y;

**Def** test(x)

printd(x) :

X = 4 :

printd(x);

test(123);

运行时，此示例打印“123”，然后打印“4”，表明我们实际上确实改变了值！

组员：

阿依多斯 2016302580021

周彪 2016302580062