编译原理研讨课：PR003

崔慧敏
cuihm@ict.ac.cn
http://www.carch.ac.cn/~huimin/main.html
助教：李奕瑾 liyijin@ict.ac.cn
李帅江 lishuaijiang@ict.ac.cn

提纲

- C语言扩展overflow check操作
- PR003要求
- PR003实现提示

提纲

- C语言扩展overflow check操作
- PR003要求
- PR003实现提示

C语言扩展overflow check操作

- 静态检查：

```
#pragma overflowCheck
int func_name(){
      int A[1000];
     int a = A[1000];
      return 0;
}
```

编译报错：

array index 1000 is past the end of the array (which contains 1000 elements)

C语言扩展overflow check操作

- 动态检查:

```
#pragma overflowCheck
int func_name(int n){
      int A[1000];
     int a = A[n];
      return 0;
}
```

C语言扩展overflow check操作

- 动态检查:

```
#pragma overflowCheck
int func_name(int n){
        int A[1000];
        assert(n < 1000 && “overflow
check failed!\n”);
        int a = A[n];
        return 0;
}
```

执行触发assert

提纲

- C语言扩展overflow check操作
- PR003要求
- PR003实现提示

PR003

- 实验内容:在C语言中为数组添加overflow check支持的优化 (仅限于`pragma overflowCheck`标注函数)
  - 死代码消除
  - 常数传播
  - 公共子表达式消除
  - 循环不变量外提
  - ……

死代码删除

- DCE: dead code elimination

  移除对程序运行结果没有任何影响的代码

- 好处
  - 避免执行不必要的操作，提高了运行的效率,减少了运行时间。
  - 节省不必要的资源分配,优化空间.
  - 减少代码的长度，增加可读性
- 例子

```
int foo(void)
{
  int a = 24;
  int b = 25; /* Assignment to dead variable */
  int c;
  c = a * 4;
  return c;
  b = 24; /* Unreachable code */
  return 0;
}
```

常量传播

- Constant propagation
- 将能够计算出结果的变量直接用常量替换
- 例子

```
int test() {
    int n = 10;
    int A[1000];
    int a = A[n];
    return 0;
}
```

```
int test() {
    int A[1000];
    int a = A[10];
    return 0;
}
```

常量折叠

- Constant Folding
- 多个变量进行计算时，而且能够直接计算出常数结果，那么变量将由常量直接替换

```
int test() {
    int A[1000];
    int n1 = 21;
    int n2 = 10;
    int n3 = n1 * n2;
    int a = A[n3];
    return 0;
}
```

常量折叠 ⟹

```
int test() {
    int A[1000];
    int n3 = 210;
    int a = A[n3];
    return 0;
}
```

常量传播 ⟹

```
int test() {
    int A[1000];
    int a = A[210];
    return 0;
}
```

循环不变量外提

- LICM: loop invariant code motion
- 将循环不变的语句或表达式移到循环体之外，而不改变程序的语义

```
int test(int n) {
    int A[1000];
    int n2;
    for (int i = 0; i < n; i++) {
        n2 = n * n;
        A[i] = i + n2;
    }
    int a = A[n2];
    return 0;
}
```

```
int test(int n) {
    int A[1000];
    int n2 = n * n;
    for (int i = 0; i < n; i++) {
        A[i] = i + n2;
    }
    int a = A[n2];
    return 0;
}
```

```
for (int i = 0; i < n; i++) {
    x = y + z;
    a[i] = 6 * i + x * x;
}
```

```
x = y + z;
t1 = x * x;
for (int i = 0; i < n; i++) {
    a[i] = 6 * i + t1;
}
```

公共子表达式消除

- CSE：common subexpression elimination
- Local CSE：工作于基本块之内
- Global CSE：工作于整个过程之中

```
int test(int n) {
    int A[1000];
    int n1 = 3 + n * n;
    int n2 = 5 + n * n;
    int a1 = A[n1];
    int a2 = A[n2];
    return 0;
}
```

```
int test(int n) {
    int A[1000];
    int n0 = n * n;
    int n1 = 3 + n0;
    int n2 = 5 + n0;
    int a1 = A[n1];
    int a2 = A[n2];
    return 0;
}
```

其他优化

- 不限

测试用例

- 一些case提示

```
int test(int n) {
    int A[1000];
    int m = 1000;
    int a = A[m];
    return 0;
}
```

常数传播
直接静态编译报错

```
int A[1000];
int test() {
    unsigned b = 1;
    int n;
    if (b) {
        n = 1000;
    } else {
        n = 100;
    }
    int a = A[n];
    return 0;
}
```

常数传播
直接静态编译报错

```
1   int A[1000];
2   int test(int n) {
3       for (int i = 0; i < n; i++) {
4           int a = A[n];
5       }
6   }
7
8
9   int A[1000];
10  int test(int n) {
11      assert(n < 1000 && "");
12      for (int i = 0; i < n; i++) {
13          int a = A[n];
14      }
15  }
```

循环不变量外提

提纲

- C语言扩展overflow check操作
- PR003要求
- PR003实现提示

LLVM Pass

- LLVM中有很多Pass用来做Analysis和Transform
- llvm/include/llvm/InitializePasses.h

```
void initializeAAEvalPass(PassRegistry&);
void initializeADCEPass(PassRegistry&);
void initializeAliasAnalysisAnalysisGroup(PassRegistry&);
void initializeAliasAnalysisCounterPass(PassRegistry&);
void initializeAliasDebuggerPass(PassRegistry&);
void initializeAliasSetPrinterPass(PassRegistry&);
void initializeAlwaysInlinerPass(PassRegistry&);
void initializeArgPromotionPass(PassRegistry&);
void initializeBarrierNoopPass(PassRegistry&);
void initializeBasicAliasAnalysisPass(PassRegistry&);
void initializeBasicCallGraphPass(PassRegistry&);
void initializeBasicTTIPass(PassRegistry&);
void initializeBlockExtractorPassPass(PassRegistry&);
void initializeBlockFrequencyInfoPass(PassRegistry&);
void initializeBlockPlacementPass(PassRegistry&);
void initializeBoundsCheckingPass(PassRegistry&);
void initializeBranchFolderPassPass(PassRegistry&);
void initializeBranchProbabilityInfoPass(PassRegistry&);
void initializeBreakCriticalEdgesPass(PassRegistry&);
void initializeCallGraphPrinterPass(PassRegistry&);
void initializeCallGraphViewerPass(PassRegistry&);
void initializeCFGOnlyPrinterPass(PassRegistry&);
void initializeCFGOnlyViewerPass(PassRegistry&);
void initializeCFGPrinterPass(PassRegistry&);
void initializeCFGSimplifyPassPass(PassRegistry&);
void initializeCFGViewerPass(PassRegistry&);
void initializeCalculateSpillWeightsPass(PassRegistry&);
void initializeCallGraphAnalysisGroup(PassRegistry&);
void initializeCodeGenPreparePass(PassRegistry&);
void initializeConstantMergePass(PassRegistry&);
void initializeConstantPropagationPass(PassRegistry&);
void initializeMachineCopyPropagationPass(PassRegistry&);
void initializeCostModelAnalysisPass(PassRegistry&);
void initializeCorrelatedValuePropagationPass(PassRegistry&);
void initializeDAEPass(PassRegistry&);
void initializeDAHPass(PassRegistry&);
```

```
void initializeDCEPass(PassRegistry&);
void initializeDSEPass(PassRegistry&);
void initializeDeadInstEliminationPass(PassRegistry&);
void initializeDeadMachineInstructionElimPass(PassRegi
void initializeDependenceAnalysisPass(PassRegistry&);
void initializeDomOnlyPrinterPass(PassRegistry&);
void initializeDomOnlyViewerPass(PassRegistry&);
void initializeDomPrinterPass(PassRegistry&);
void initializeDomViewerPass(PassRegistry&);
void initializeDominanceFrontierPass(PassRegistry&);
void initializeDominatorTreePass(PassRegistry&);
void initializeEarlyIfConverterPass(PassRegistry&);
void initializeEdgeBundlesPass(PassRegistry&);
void initializeEdgeProfilerPass(PassRegistry&);
void initializeExpandPostRAPass(PassRegistry&);
void initializePathProfilerPass(PassRegistry&);
void initializeGCOVProfilerPass(PassRegistry&);
void initializeAddressSanitizerPass(PassRegistry&);
void initializeAddressSanitizerModulePass(PassRegistry
void initializeMemorySanitizerPass(PassRegistry&);
void initializeThreadSanitizerPass(PassRegistry&);
void initializeEarlyCSEPass(PassRegistry&);
void initializeExpandISelPseudosPass(PassRegistry&);
void initializeFindUsedTypesPass(PassRegistry&);
void initializeFunctionAttrsPass(PassRegistry&);
void initializeGCMachineCodeAnalysisPass(PassRegistry&
void initializeGCModuleInfoPass(PassRegistry&);
void initializeGVNPass(PassRegistry&);
void initializeGlobalDCEPass(PassRegistry&);
void initializeGlobalOptPass(PassRegistry&);
void initializeGlobalsModRefPass(PassRegistry&);
void initializeIPCPPass(PassRegistry&);
void initializeIPSCCPPass(PassRegistry&);
void initializeIVUsersPass(PassRegistry&);
void initializeIfConverterPass(PassRegistry&);
void initializeIndVarSimplifyPass(PassRegistry&);
void initializeInlineCostAnalysisPass(PassRegistry&);
```

LLVM Pass

```
void initializeInstCombinerPass(PassRegistry&);
void initializeInstCountPass(PassRegistry&);
void initializeInstNamerPass(PassRegistry&);
void initializeInternalizePassPass(PassRegistry&);
void initializeIntervalPartitionPass(PassRegistry&);
void initializeJumpThreadingPass(PassRegistry&);
void initializeLCSSAPass(PassRegistry&);
void initializeLICMPass(PassRegistry&);
void initializeLazyValueInfoPass(PassRegistry&);
void initializeLibCallAliasAnalysisPass(PassRegistry&);
void initializeLintPass(PassRegistry&);
void initializeLiveDebugVariablesPass(PassRegistry&);
void initializeLiveIntervalsPass(PassRegistry&);
void initializeLiveRegMatrixPass(PassRegistry&);
void initializeLiveStacksPass(PassRegistry&);
void initializeLiveVariablesPass(PassRegistry&);
void initializeLoaderPassPass(PassRegistry&);
void initializeProfileMetadataLoaderPassPass(PassRegistry&);
void initializePathProfileLoaderPassPass(PassRegistry&);
void initializeLocalStackSlotPassPass(PassRegistry&);
void initializeLoopDeletionPass(PassRegistry&);
void initializeLoopExtractorPass(PassRegistry&);
void initializeLoopInfoPass(PassRegistry&);
void initializeLoopInstSimplifyPass(PassRegistry&);
void initializeLoopRotatePass(PassRegistry&);
void initializeLoopSimplifyPass(PassRegistry&);
void initializeLoopStrengthReducePass(PassRegistry&);
void initializeGlobalMergePass(PassRegistry&);
void initializeLoopUnrollPass(PassRegistry&);
void initializeLoopUnswitchPass(PassRegistry&);
void initializeLoopIdiomRecognizePass(PassRegistry&);
void initializeLowerAtomicPass(PassRegistry&);
void initializeLowerExpectIntrinsicPass(PassRegistry&);
void initializeLowerIntrinsicsPass(PassRegistry&);
void initializeLowerInvokePass(PassRegistry&);
void initializeLowerSwitchPass(PassRegistry&);
```

```
void initializeMachineBranchProbabilityInfoPass(PassRegistry&);
void initializeMachineCSEPass(PassRegistry&);
void initializeMachineDominatorTreePass(PassRegistry&);
void initializeMachinePostDominatorTreePass(PassRegistry&);
void initializeMachineLICMPass(PassRegistry&);
void initializeMachineLoopInfoPass(PassRegistry&);
void initializeMachineModuleInfoPass(PassRegistry&);
void initializeMachineSchedulerPass(PassRegistry&);
void initializeMachineSinkingPass(PassRegistry&);
void initializeMachineTraceMetricsPass(PassRegistry&);
void initializeMachineVerifierPassPass(PassRegistry&);
void initializeMemCpyOptPass(PassRegistry&);
void initializeMemDepPrinterPass(PassRegistry&);
void initializeMemoryDependenceAnalysisPass(PassRegistry&);
void initializeMetaRenamerPass(PassRegistry&);
void initializeMergeFunctionsPass(PassRegistry&);
void initializeModuleDebugInfoPrinterPass(PassRegistry&);
void initializeNoAAPass(PassRegistry&);
void initializeNoProfileInfoPass(PassRegistry&);
void initializeNoPathProfileInfoPass(PassRegistry&);
void initializeObjCARCAliasAnalysisPass(PassRegistry&);
void initializeObjCARCAPElimPass(PassRegistry&);
void initializeObjCARCExpandPass(PassRegistry&);
void initializeObjCARCContractPass(PassRegistry&);
void initializeObjCARCOptPass(PassRegistry&);
void initializeOptimalEdgeProfilerPass(PassRegistry&);
void initializeOptimizePHIsPass(PassRegistry&);
void initializePEIPass(PassRegistry&);
void initializePHIEliminationPass(PassRegistry&);
void initializePartialInlinerPass(PassRegistry&);
void initializePeepholeOptimizerPass(PassRegistry&);
void initializePostDomOnlyPrinterPass(PassRegistry&);
void initializePostDomOnlyViewerPass(PassRegistry&);
void initializePostDomPrinterPass(PassRegistry&);
void initializePostDomViewerPass(PassRegistry&);
void initializePostDominatorTreePass(PassRegistry&);
void initializePostRASchedulerPass(PassRegistry&);
```

LLVM Pass

- Each pass is one analysis or transformation
- Five types of Pass

  ModulePass: Use the entire program as a unit

  CallGraphSCCPass: Traverse the program bottom-up on the call graph

  FunctionPass: Execute on each function in the program

  LoopPass： Process loops in loop nest order

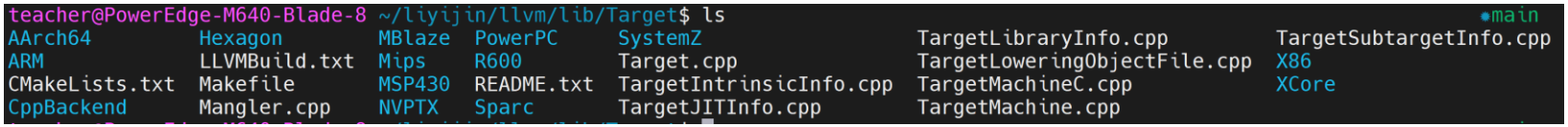
  RegionPass: execute on each single entry single exit region

LLVM Pass

- Analysis Pass
- Transform Pass

  不同的层级，如DCE和MachineDCE

  前者在LLVM IR上，后者在Machine Inst上

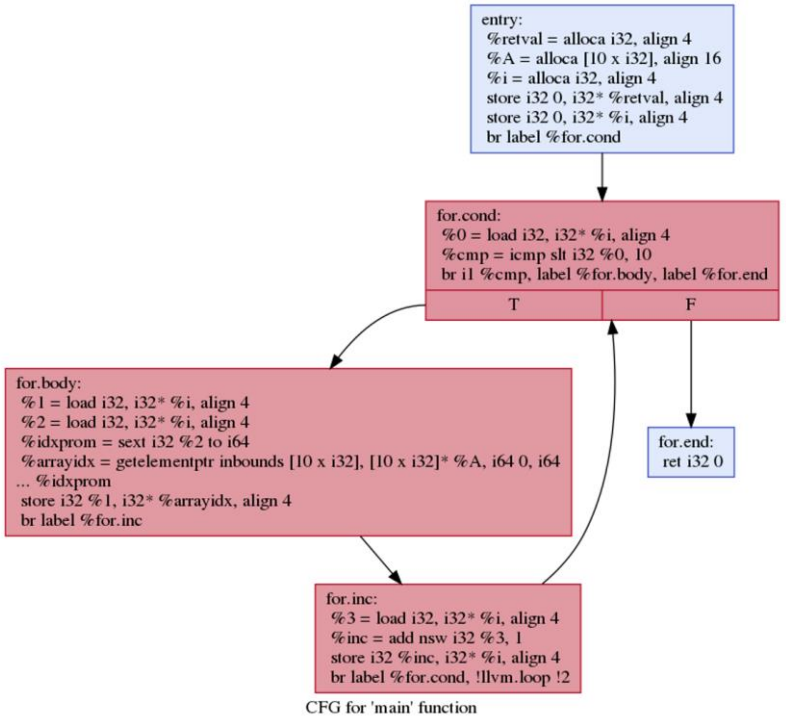


- Printer Pass：for debug
- ……

Control Flow Graph

- CFG：用图的形式表示一个过程内所有基本块执行的可能流向
- LLVM IR由于其语言特性，其BasicBlock已经以CFG的结构组织

```
; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
entry:
  %retval = alloca i32, align 4
  %A = alloca [10 x i32], align 16
  %i = alloca i32, align 4
  store i32 0, i32* %retval, align 4
  store i32 0, i32* %i, align 4
  br label %for.cond

for.cond:                                         ; preds = %for.inc, %entry
  %0 = load i32, i32* %i, align 4
  %cmp = icmp slt i32 %0, 10
  br i1 %cmp, label %for.body, label %for.end

for.body:                                         ; preds = %for.cond
  %1 = load i32, i32* %i, align 4
  %2 = load i32, i32* %i, align 4
  %idxprom = sext i32 %2 to i64
  %arrayidx = getelementptr inbounds [10 x i32], [10 x i32]* %A, i64 0, i64 %idxprom
  store i32 %1, i32* %arrayidx, align 4
  br label %for.inc

for.inc:                                          ; preds = %for.body
  %3 = load i32, i32* %i, align 4
  %inc = add nsw i32 %3, 1
  store i32 %inc, i32* %i, align 4
  br label %for.cond, !llvm.loop !2

for.end:                                          ; preds = %for.cond
  ret i32 0
}
```



CFG for 'main' function

SSA

- LLVM IR是SSA形式
  - 静态单赋值static single assignment
  - 每个变量只定值一次
- 简化程序中变量
  - 简化编译优化方法
  - 可以获得更好的优化结果

```
y := 1
y := 2
x := y
```

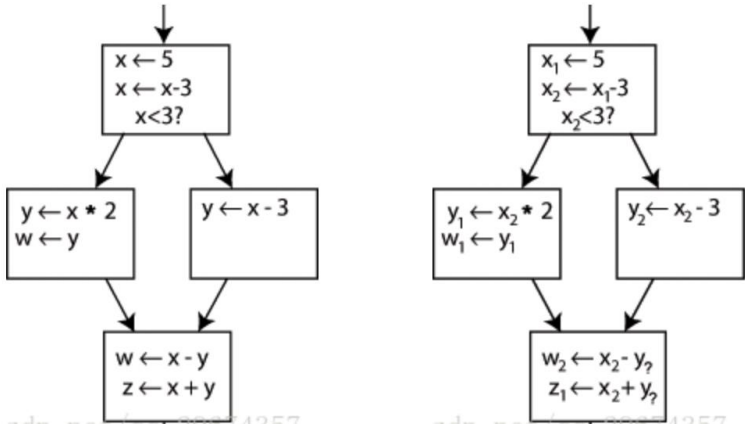
```
y1 := 1
y2 := 2
x1 := y2
```

第三行使用的y值来自于第二行的赋值。
对于非SSA IR的编译来说，需要做到达-定义分析，确定选取哪一行的y值。
但是对于SSA IR来说，不需要做分析即可知道

SSA

- 如何生成SSA?

  将每个被赋值变量用一个新的版本代替，并将使用的变量替换为该变量到达该程序点的版本



- 最后一个BB的y来自于?
- phi结点

SSA

```
int test(int* a) {
  for (int i = 0; i < 10; i++) {
     *(a+i) = i;
}
return 0;
}
```

- clang -S a.c -emit-llvm -O1

```
; Function Attrs: nounwind uwtable
define i32 @test(i32* nocapture %a) #0 {
entry:
  br label %for.body

for.body:                                         ; preds = %for.body, %entry
  %indvars.iv = phi i64 [ 0, %entry ], [ %indvars.iv.next, %for.body ]
  %add.ptr = getelementptr inbounds i32* %a, i64 %indvars.iv
  %0 = trunc i64 %indvars.iv to i32
  store i32 %0, i32* %add.ptr, align 4, !tbaa !0
  %indvars.iv.next = add i64 %indvars.iv, 1
  %lftr.wideiv = trunc i64 %indvars.iv.next to i32
  %exitcond = icmp eq i32 %lftr.wideiv, 10
  br i1 %exitcond, label %for.end, label %for.body

for.end:                                          ; preds = %for.body
  ret i32 0
}
```

- phi指令根据在当前bb之前执行的是哪一个predecessor bb来得到相应的值
- 用-O0编译，看看有什么区别？

Mem2Reg

- lib/Transforms/Utils/Mem2Reg
- 将memory访问提升为寄存器访问
- LLVM假设程序中所有的局部变量都在栈上，并且通过Alloca指令在函数的Entry BB进行声明，并且只出现在Entry BB里
- 练习： opt -S a.s -mem2reg -o b.s

```
int test() {
  int n = 1;
  int j = 2;
  int k = 3;
  int m = n * j + k;
  return 0;
}
```

```
define i32 @test() #0 {
entry:
  %n = alloca i32, align 4
  %j = alloca i32, align 4
  %k = alloca i32, align 4
  %m = alloca i32, align 4
  store i32 1, i32* %n, align 4
  store i32 2, i32* %j, align 4
  store i32 3, i32* %k, align 4
  %0 = load i32* %n, align 4
  %1 = load i32* %j, align 4
  %mul = mul nsw i32 %0, %1
  %2 = load i32* %k, align 4
  %add = add nsw i32 %mul, %2
  store i32 %add, i32* %m, align 4
  ret i32 0
}
```

```
define i32 @test() #0 {
entry:
  %mul = mul nsw i32 1, 2
  %add = add nsw i32 %mul, 3
  ret i32 0
}
```

Dominance

- 支配：

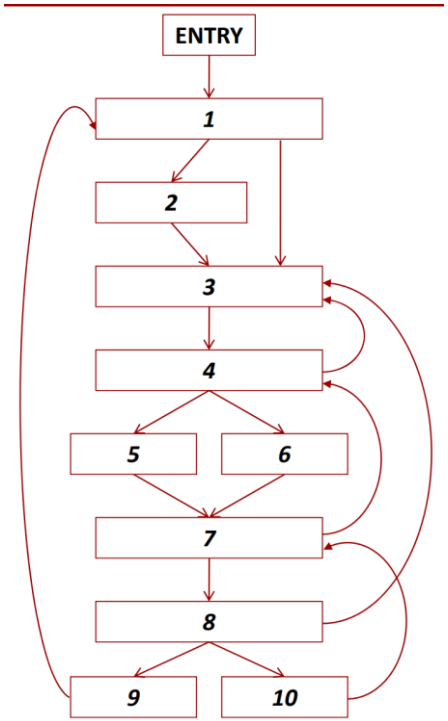
  如果每一条从流图的入口结点到结点n的路径都经过结点d，则称结点d支配（dominate）结点n，记作d dom n（每个结点都支配自己）

- 严格支配 strictly dominate

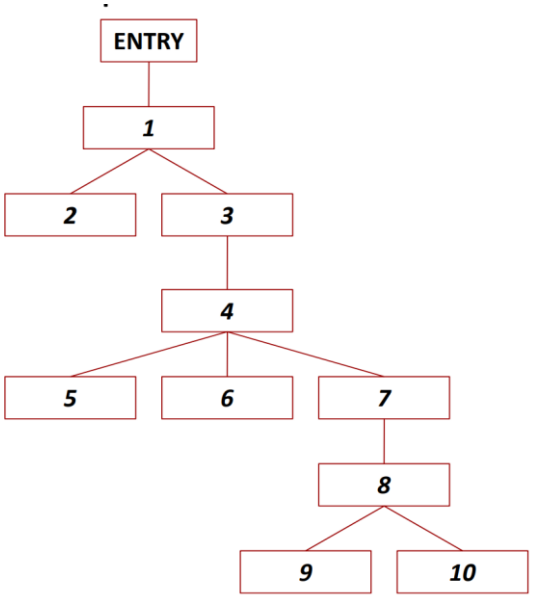
  如果d != n 且 d dom n，那么d sdom n

Dominator Tree

- 支配树（dominator tree）用来表示支配信息。在树中，每个结点只支配它在树中的后代结点。



CFG



Dominator Tree

Dominator Tree

- lib/IR/Dominators.cpp,构建了DominatorTree
- 练习： opt -dot-dom a.s

  生成 dom.xx.dot文件

  dot –Tpng –o test.png dom.xx.dot
- 接口

Public Member Functions

	DominatorTree ()=default
	DominatorTree (**Function** &F)
	DominatorTree (**DominatorTree** &DT, **DomTreeBuilder::BBUpdates** U)
bool	**invalidate** (**Function** &F, **const PreservedAnalyses** &PA, **FunctionAnalysisManager::Invalidator** & Handle invalidation explicitly. More...
bool	**dominates** (**const BasicBlock** *BB, **const Use** &U) **const** Return true if the (end of the) basic block BB dominates the use U. More...
bool	**dominates** (**const Value** *Def, **const Use** &U) **const** Return true if value Def dominates use U, in the sense that Def is available at U, and could be substi
bool	**dominates** (**const Value** *Def, **const Instruction** *User) **const** Return true if value Def dominates all possible uses inside instruction **User**. More...
bool	**dominates** (**const Instruction** *Def, **const BasicBlock** *BB) **const**
bool	**dominates** (**const BasicBlockEdge** &BBE, **const Use** &U) **const** Return true if an edge dominates a use. More...
bool	**dominates** (**const BasicBlockEdge** &BBE, **const BasicBlock** *BB) **const**
bool	**dominates** (**const BasicBlockEdge** &BBE1, **const BasicBlockEdge** &BBE2) **const** Returns true if edge BBE1 dominates edge BBE2. More...
bool	**isReachableFromEntry** (**const Use** &U) **const** Provide an overload for a **Use**. More...
void	**viewGraph** (**const Twine** &Name, **const Twine** &Title)
void	**viewGraph** ()

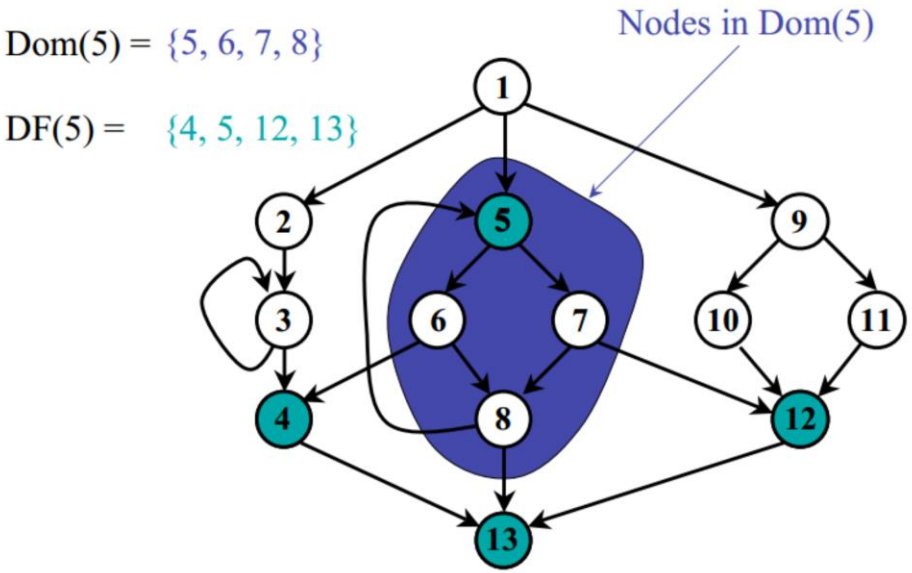
```
digraph "Dominator tree for 'test' function" {
    label="Dominator tree for 'test' function";

    Node0x2b012c0 [shape=record,label="{entry:\l  br label %for.body\l}"];
    Node0x2b012c0 -> Node0x2b01300;
    Node0x2b01300 [shape=record,label="{for.body:
y ], [ %indvars.iv.next, %for.body ]\l  %add.ptr = getelementptr inbounds i3
iv to i32\l  store i32 %0, i32* %add.ptr, align 4, !tbaa !0\l  %indvars.iv.n
nc i64 %indvars.iv.next to i32\l  %exitcond = icmp eq i32 %lftr.wideiv, 10\l
\l}"];
    Node0x2b01300 -> Node0x2b01340;
    Node0x2b01340 [shape=record,label="{for.end:
}
```

Dominance Frontier

- 支配边界Dominance Frontier

  直观理解：当前结点所能支配的边界

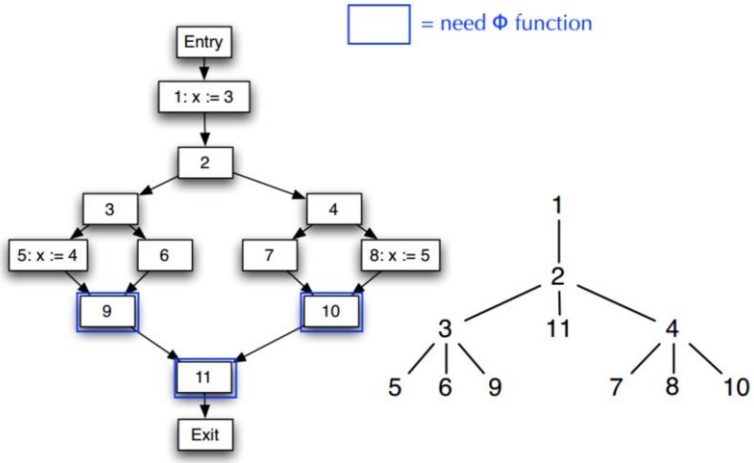
  定义描述：Y是X的支配边界，当且仅当X支配Y的一个前驱结点（CFG），同时X并不严格支配Y



结点5刚好不能支配的地方

Dominance Frontier

- lib/Analysis/DominanceFrontier.cpp
- 支配边界确定了phi-函数的插入位置
- 由于每个def支配对应的use，所以如果达到了def所在bb的支配边界，就必须考虑其他路径是否有相同variable的定义，由于在编译期间无法确定会采用哪一分支，所以需要放置phi-函数



Post Dominate

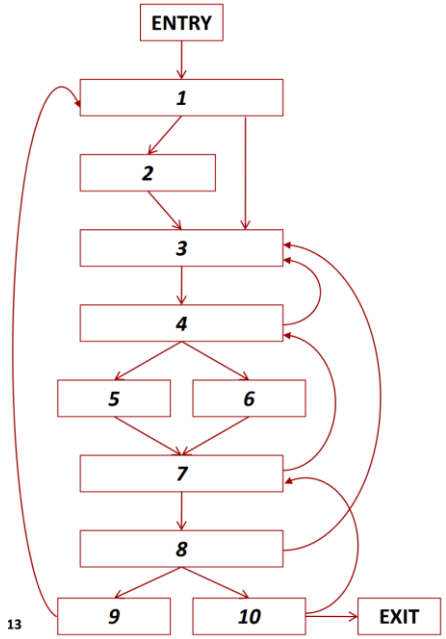
- Post-dominate：如果CFG中的结点n到出口的所有路径都经过结点z，则z pdom n
- 结点m是n的immediate post-dominator
- CFG上的post-dominance等价于相反的CFG上的dominance（所有边反向）

Public Member Functions

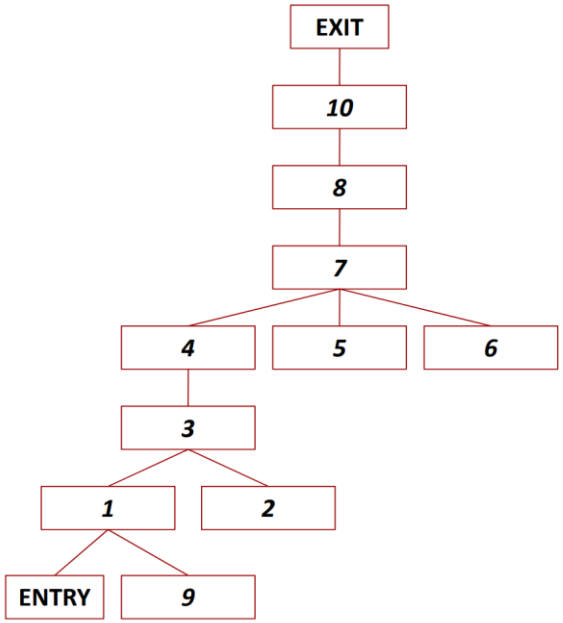
	PostDominatorTree ()=default
	PostDominatorTree (**Function** &**F**)
bool	**invalidate** (**Function** &**F**, **const PreservedAnalyses** &PA, **FunctionAnalysisManager::Invalidator** &) Handle invalidation explicitly. More...
bool	**dominates** (**const Instruction** *I1, **const Instruction** *I2) **const** Return true if I1 dominates I2. More...
bool	**dominates** (**const DomTreeNodeBase**< NodeT > *A, **const DomTreeNodeBase**< NodeT > ***B**) **const** dominates - Returns true iff A dominates B. More...
bool	**dominates** (**const** NodeT *A, **const** NodeT ***B**) **const**

Post Dominator Tree

- n的父亲是它的直接post-dominator，根结点是Exit
- 练习： opt -dot-postdom a.s



CFG



Post Dominator Tree

Alias Analysis

- 判断两个指针是否指向同一个memory object
- 许多AA算法,分类:

  flow-sensitive vs flow-insensitive

  path-sensitive vs path-insensitive

  context-sensitive vs context-insensitive

  field-sensitive vs field-insensitive

Alias Analysis

- 在flow-insensitive的指针分析中，计算整个程序的指向集合。stmt的order不重要，S1;S2的结果和S2;S1一样。

  pts(p)={a,b}

- 在flow-sensitive的指针分析中，每个stmt之后会计算可能的指针值。

  在程序点4，pts(p)={a}, 在程序点5,pts(p)={b}

- 后者更加precise，但是less efficient，too expensivie for whole program。

```
1 int a;
2 int b;
3 int* p;
4 p = &a;
5 p = &b;
```

Listing 2.5: Example of flow-sensitivity

Alias Analysis

- path-insensitive：不区分路径

  pts(p)={a,b,c,d} imprecise

- path-sensitive：区分路径

  p值可能指向c或者指向d,两者不可能同时存在

```
1  int a;
2  int b;
3  int c;
4  int* p;
5  p = &a
6  p = &b;
7  if (s){
8      p = &c;
9  }
10 else {
11     p = &d;
12 }
```

Listing 2.6: Example of path-sensitivity

Alias Analysis

- context-insensitive：不区分函数调用上下文

  id的返回可能是1或2，则c的值可能是2，3，4.

- context-sensitive:区分函数调用上下文

  区分id(1)和id(2)，c=3

replacing a+b by 3.

```
1 int a = id(1);
2 int b = id(2);
3 int c = a + b;
4
5 int id(int x){
6     return x;
7 }
```

Listing 2.7: Example of context-sensitivity

Alias Analysis

- field-insensitive:将struct看作一个整体
- field-sensitive： 将struct的成员变量看成单独的元素

```
1 struct point {
2     int x;
3     int y;
4 }
5 struct point p1, p2;
6 int * a, b, c;
7 p1.x = &a;
8 p1.y = &b;
9 p2.x = &c;
```

Listing 2.8: Example of field-sensitivity

A field-sensitive analysis would yield the precise result that `pts(p1.x)={a}`, `pts(p1.y)={b}` and `pts(p2.x)={c}`. A field-insensitive analysis would not distinguish between the different fields of the struct. It would, therefore, conclude that `pts(p1)={a, b}` and `pts(p2)={c}`. On the other hand, a field-based analysis distinguishes between the fields of the struct but is not able to tell apart different instances of the struct and would therefore yield `pts(x)={a,c}` and `pts(y)={b}`.

Alias Analysis

- AA的两个基础算法
- Andersen：inclusion-based，基于subset constraints

  如果a=b，则pts(b) ⊆ pts(a)
- Steensgaard：unification-based，基于equality constaints

  将子集关系换成等号，从而使得指针分析化简

  如果a=b，则pts(b) = pts(a)
- 推荐阅读：

  南京大学《软件分析》课程，李樾 谭添

  https://pascal-group.bitbucket.io/teaching.html

Alias Analysis

- AliasAnalysis(AAResults) class是LLVM AA中的主要接口
  - 定义了不同AA实现应该支持的接口
  - 提供方法用来query两个momory objects是否alias，函数调用是否读或写memory object等
  - 该类包含两个enums类: AliasResult和ModRefResult，分别代表了alias query 和 mod/ref query的结果
  - Memory object用MemoryLocation类描述，该类是对specific location in memory的抽象，包含start address和size
  - lib/Analysis/AliasAnalysis.cpp

Alias Analysis

- MemoryLocation class表示memory location

  Address of location start，Size

- AliasResult class表示alias query的结果

  {No，May，Partial，Must}alias

- ModRefInfo class表示mod/ref query的结果

  {Must, MustRef, MustMod, MustModRef, NoModRef, Ref, Mod, ModRef}

```
/// The address of the start of the location.
const Value *Ptr;

/// The maximum size of the location, in address-units, or
/// UnknownSize if the size is not known.
///
/// Note that an unknown size does not mean the pointer aliases the entire
/// virtual address space, because there are restrictions on stepping out of
/// one object and into another. See
/// http://llvm.org/docs/LangRef.html#pointeraliasing
LocationSize Size;
```

```
enum Kind : uint8_t {
  /// The two locations do not alias at all.
  ///
  /// This value is arranged to convert to false, while all other values
  /// convert to true. This allows a boolean context to convert the result to
  /// a binary flag indicating whether there is the possibility of aliasing.
  NoAlias = 0,
  /// The two locations may or may not alias. This is the least precise
  /// result.
  MayAlias,
  /// The two locations alias, but only due to a partial overlap.
  PartialAlias,
  /// The two locations precisely alias each other.
  MustAlias,
};
```

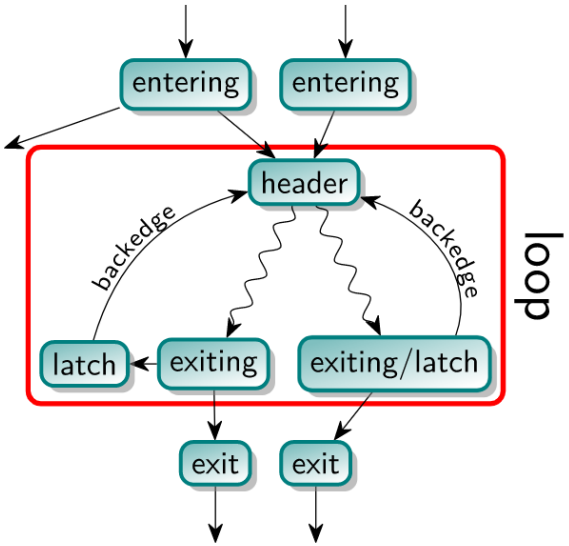
```
enum class ModRefInfo : uint8_t {
  /// Must is provided for completeness, but no routines will return only
  /// Must today. See definition of Must below.
  Must = 0,
  /// The access may reference the value stored in memory,
  /// a mustAlias relation was found, and no mayAlias or partialAlias found.
  MustRef = 1,
  /// The access may modify the value stored in memory,
  /// a mustAlias relation was found, and no mayAlias or partialAlias found.
  MustMod = 2,
  /// The access may reference, modify or both the value stored in memory,
  /// a mustAlias relation was found, and no mayAlias or partialAlias found.
  MustModRef = MustRef | MustMod,
  /// The access neither references nor modifies the value stored in memory.
  NoModRef = 4,
  /// The access may reference the value stored in memory.
  Ref = NoModRef | MustRef,
  /// The access may modify the value stored in memory.
  Mod = NoModRef | MustMod,
  /// The access may reference and may modify the value stored in memory.
  ModRef = Ref | Mod,
```

Alias Analysis

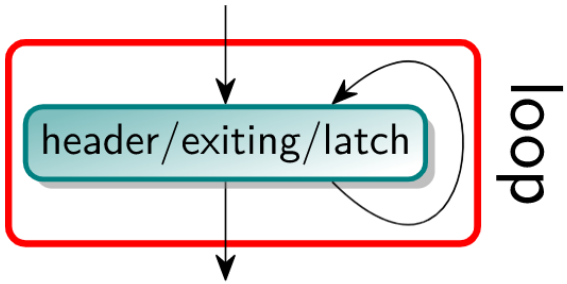
- LLVM实现的不同的AA方法：
- basic-aa： Basic Alias Analysis
- globalsmodref-aa: Globals Alias Analysis
- cfl-anders-aa： Inclusion-Based CFL Alias Analysis
- cfl-steens-aa: Unification-Based CFL Alias Analysis
- external-aa: External Alias Analysis
- Scev-aa: ScalarEvolution-based Alias Analysis
- Tbaa: Type-Based Alias Analysis
- ……

Loop Analysis

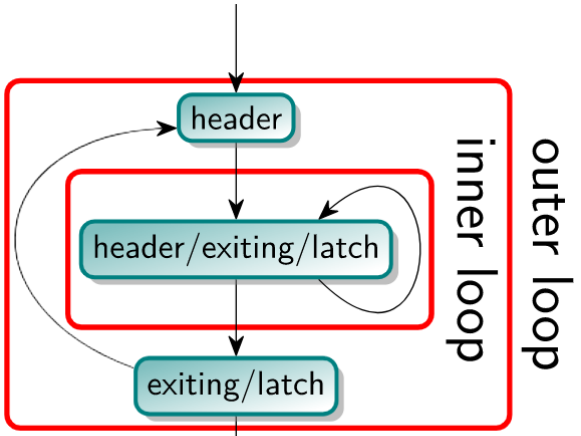
- Loop是代码优化的重要概念
- LLVM中，CFG中的loops是通过LoopInfo检测的
- Loop由entering block，header，latch，backedge等组成



Loop



最小loop



nested loop

Loop Analysis

- Loop class表示CFG中的a single loop,该类内的：
  LocRange class，表示loop的start和end location
  LoopBounds class，表示loop的上下界
  getInductionVariable，用来得到loop的induction variable

```
/// A range representing the start and end location of a loop.
class LocRange {
  DebugLoc Start;
  DebugLoc End;

public:
  LocRange() = default;
  LocRange(DebugLoc Start) : Start(Start), End(Start) {}
  LocRange(DebugLoc Start, DebugLoc End)
      : Start(std::move(Start)), End(std::move(End)) {}

  const DebugLoc &getStart() const { return Start; }
  const DebugLoc &getEnd() const { return End; }

  /// Check for null.
  ///
  explicit operator bool() const { return Start && End; }
};
```

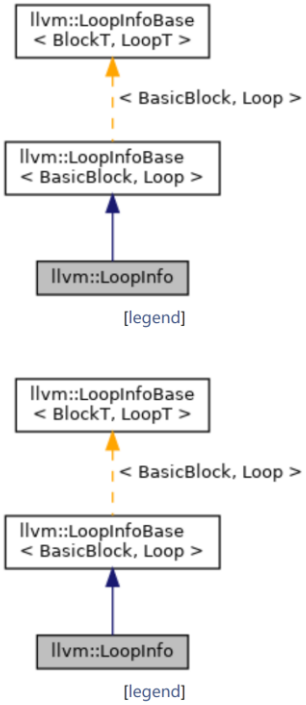
LocRange定义

```
struct LoopBounds {
  /// Return the LoopBounds object if
  /// - the given \p IndVar is an induction variable
  /// - the initial value of the induction variable can be found
  /// - the step instruction of the induction variable can be found
  /// - the final value of the induction variable can be found
  ///
  /// Else None.
  static Optional<Loop::LoopBounds> getBounds(const Loop &L, PHINode &IndVar,
                                              ScalarEvolution &SE);

  /// Get the initial value of the loop induction variable.
  Value &getInitialIVValue() const { return InitialIVValue; }

  /// Get the instruction that updates the loop induction variable.
  Instruction &getStepInst() const { return StepInst; }

  /// Get the step that the loop induction variable gets updated by in each
  /// loop iteration. Return nullptr if not found.
  Value *getStepValue() const { return StepValue; }

  /// Get the final value of the loop induction variable.
  Value &getFinalIVValue() const { return FinalIVValue; }
```

LoopBounds定义

Loop Analysis

- LoopInfo是获得loop相关信息的core analysis
- 能够通过LoopInfo遍历所有的Loop(top level and sub-tree)

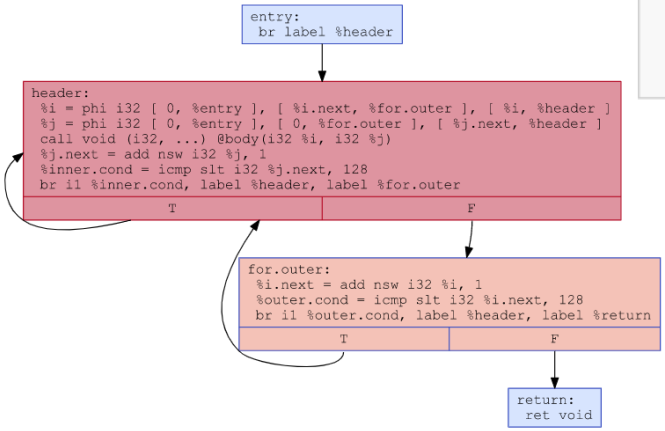


SmallVector< Loop *, 4 >	**getLoopsInPreorder** () const Return all of the loops in the function in preorder
SmallVector< Loop *, 4 >	**getLoopsInReverseSiblingPreorder** () const Return all of the loops in the function in preorder
Loop *	**getLoopFor** (const BasicBlock *BB) const Return the inner most loop that BB lives in. More.
const Loop *	**operator[]** (const BasicBlock *BB) const Same as getLoopFor. More...
unsigned	**getLoopDepth** (const BasicBlock *BB) const Return the loop nesting level of the specified bloc
bool	**isLoopHeader** (const BasicBlock *BB) const
const std::vector< Loop * > &	**getTopLevelLoops** () const Return the top-level loops. More...
std::vector< Loop * > &	**getTopLevelLoopsVector** () Return the top-level loops. More...
Loop *	**removeLoop** (iterator I) This removes the specified top-level loop from th

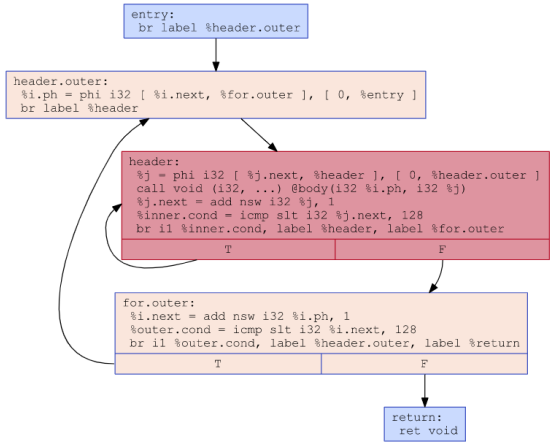
Loop Analysis

- Loop Simplify Form:将Loop变为canonical form，能够使analysis和transformation更简单高效
- 当调度LoopPass时LoopSimplify Pass会被自动加入pass manager中
- 该pass后，loop形式为：包含a preheader，a single backedge，dedicated exits

```
for (int i = 0; i < 128; ++i)
  for (int j = 0; j < 128; ++j)
    body(i,j);
```



LoopSimplify



1个preheader

为nested loop分开preheader

Dependence Analysis

- LLVM分析memory access的dependence
- 可以query 两个指令的依赖关系

Input：

- 变量在一个表达式中被使用，在后来另一表达式中被使用

Output dep:

- 变量在一个表达式中被修改赋值，在后来另一表达式中被修改赋值

Flow dep:

- 变量在一个表达式中被修改赋值，在后来另一表达式被使用

Anti dep：

- 变量在一个表达式中被使用，在后来另一个表示式中被修改赋值

Dependence Analysis

- lib/Analysis/DependenceAnalysis.cpp
- 接口

```
static
void dumpExampleDependence(raw_ostream &OS, Function *F,
                           DependenceAnalysis *DA) {
  for (inst_iterator SrcI = inst_begin(F), SrcE = inst_end(F);
       SrcI != SrcE; ++SrcI) {
    if (isa<StoreInst>(*SrcI) || isa<LoadInst>(*SrcI)) {
      for (inst_iterator DstI = SrcI, DstE = inst_end(F);
           DstI != DstE; ++DstI) {
        if (isa<StoreInst>(*DstI) || isa<LoadInst>(*DstI)) {
          OS << "da analyze - ";
          if (Dependence *D = DA->depends(&*SrcI, &*DstI, true)) {
            D->dump(OS);
            for (unsigned Level = 1; Level <= D->getLevels(); Level++) {
              if (D->isSplitable(Level)) {
                OS << "da analyze - split level = " << Level;
                OS << ", iteration = " << *DA->getSplitIteration(D, Level);
                OS << "!\n";
              }
            }
            delete D;
          }
          else
            OS << "none!\n";
        }
      }
    }
  }
}
```

Scalar Evolution

- 用来分析loop中induction variable的表达式

  标量表达式的表示：不同类型的表达式表示为SCEV类的不同子类

  - SCEVConstant, SCEVCastExpr, SCEVAddExpr, SCEVMulExpr, SCEVDivExpr, SCEVAddRecExpr, SCEVUnknown,……

  Canonical form：能比较两个SCEV指针是否相等

```
enum SCEVTypes : unsigned short {
  // These should be ordered in term
  // folders simpler.
  scConstant,
  scTruncate,
  scZeroExtend,
  scSignExtend,
  scAddExpr,
  scMulExpr,
  scUDivExpr,
  scAddRecExpr,
  scUMaxExpr,
  scSMaxExpr,
  scUMinExpr,
  scSMinExpr,
  scSequentialUMinExpr,
  scPtrToInt,
  scUnknown,
  scCouldNotCompute
};
```

Scalar Evolution

- SCEVAddRecExpr 是ScalarEvolution的核心
- 表示polynomial recurrence on the trip count of the specified loop
- 其他的SCEV子类主要是为了支持创建分析SCEVAddRecExpr
- 形式为： recurrence = {start，+，step}

Type *	**getType** () **const**
const SCEV *	**getStart** () **const**
const Loop *	**getLoop** () **const**
const SCEV *	**getStepRecurrence** (**ScalarEvolution** &SE) **const** Constructs and returns the recurrence indicating how much this expression steps by. More...
bool	**isAffine** () **const** Return true if this represents an expression A + B*x where A and B are loop invariant values. More...
bool	**isQuadratic** () **const** Return true if this represents an expression A + B*x + C*x^2 where A, B and C are loop invariant values.
void	**setNoWrapFlags** (**NoWrapFlags** Flags) Set flags for a recurrence without clearing any previously set flags. More...
const SCEV *	**evaluateAtIteration** (**const SCEV** *It, **ScalarEvolution** &SE) **const** Return the value of this chain of recurrences at the specified iteration number. More...
const SCEV *	**getNumIterationsInRange** (**const ConstantRange** &Range, **ScalarEvolution** &SE) **const** Return the number of iterations of this loop that produce values in the specified constant range. More...
const SCEVAddRecExpr *	**getPostIncExpr** (**ScalarEvolution** &SE) **const** Return an expression representing the value of this expression one iteration of the loop ahead. More...

Constant Propagation

- llvm/lib/Transforms/Scalar/ConstantProp.cpp
- FunctionPass
- Pass的注册方式在llvm的新版本中有变化(LLVM 12)

```
namespace {
  struct ConstantPropagation : public FunctionPass {
    static char ID; // Pass identification, replacement for typeid
    ConstantPropagation() : FunctionPass(ID) {
      initializeConstantPropagationPass(*PassRegistry::getPassRegistry());
    }

    bool runOnFunction(Function &F);

    virtual void getAnalysisUsage(AnalysisUsage &AU) const {
      AU.setPreservesCFG();
      AU.addRequired<TargetLibraryInfo>();
    }
  };
}
```

Constant Propagation

- runOnFunction实现，主要实现逻辑

```
bool ConstantPropagation::runOnFunction(Function &F) {
  // Initialize the worklist to all of the instructions ready to process...
  std::set<Instruction*> WorkList;
  for(inst_iterator i = inst_begin(F), e = inst_end(F); i != e; ++i) {
    WorkList.insert(&*i);
  }
  bool Changed = false;
  DataLayout *TD = getAnalysisIfAvailable<DataLayout>();
  TargetLibraryInfo *TLI = &getAnalysis<TargetLibraryInfo>();

  while (!WorkList.empty()) {
    Instruction *I = *WorkList.begin();
    WorkList.erase(WorkList.begin());    // Get an element from the worklist...

    if (!I->use_empty())                 // Don't muck with dead instructions...
      if (Constant *C = ConstantFoldInstruction(I, TD, TLI)) {
        // Add all of the users of this instruction to the worklist, they might
        // be constant propagatable now...
        for (Value::use_iterator UI = I->use_begin(), UE = I->use_end();
             UI != UE; ++UI)
          WorkList.insert(cast<Instruction>(*UI));

        // Replace all of the uses of a variable with uses of the constant.
        I->replaceAllUsesWith(C);

        // Remove the dead instruction.
        WorkList.erase(I);
        I->eraseFromParent();

        // We made a change to the function...
        Changed = true;
        ++NumInstKilled;
      }
  }
  return Changed;
}
```

将Function的所有指令放到workList中

遍历worklist处理每条指令

只处理!use_empty()的指令– 死代码处理无用

将当前指令能够成功折叠成Constant
此处逻辑复杂，见该函数实现

遍历指令的use

用该constant替换掉I的使用

需要回答：为什么返回Changed?

CSE

- 可用表达式

  表达式x+y在点p可用：如果从初始结点到p的任意路径上都计算x+y，且在最后一个这样的计算和p之间没有对x或y的赋值

语　句	可用表达式
……	
a := b + c	
……	
b := a - d	
……	
c := b + c	
……	
d := a - d	
……	

语　句	可用表达式
……	无
a := b + c	
……	产生b + c　{b+c}
b := a - d	
……	注销b+c　产生a – d　{a-d}
c := b + c	
……	产生b+c，仅有a – d　{a-d}
d := a - d	{a-d,b+c}
……	无　{}

CSE

- 相关的集合定义：对于每个基本块B，U是程序中出现在语句右部的所有表达式集合
  - in[B]：块B开始点的可用表达式集合
  - out[B]：块B结束点的可用表达式集合
  - e_gen[B]：块B生成的可用表达式集合
  - e_kill[B]：U中被块B注销的可用表达式集合

- 数据流方程（正向数据流分析）

$$out[B] = (in[B] - e\_kill[B]) \cup e\_gen[B]$$

$$in[B] = \bigcap_{P\text{是 }B\text{的前驱}} out[P] \quad (\text{B不是开始块})$$

$$in[B1] = \varnothing \quad (\text{B1是开始块})$$

CSE

- LLVM不是对基础算法的简单实现
- Early CSE
- A simple and fast domtree-based CSE pass.
- This pass does a simple depth-first walk over the dominator tree, eliminating trivially redundant instructions and using instsimplify to canonicalize things as it goes.

CSE

- lib/Transforms/Scalar/EarlyCSE.cpp

```
bool EarlyCSE::runOnFunction(Function &F) {
  std::deque<StackNode *> nodesToProcess;

  TD = getAnalysisIfAvailable<DataLayout>();
  TLI = &getAnalysis<TargetLibraryInfo>();
  DT = &getAnalysis<DominatorTree>();

  // Tables that the pass uses when walking the domtree.
  ScopedHTType AVTable;
  AvailableValues = &AVTable;
  LoadHTType LoadTable;
  AvailableLoads = &LoadTable;
  CallHTType CallTable;
  AvailableCalls = &CallTable;

  CurrentGeneration = 0;
  bool Changed = false;

  // Process the root node.
  nodesToProcess.push_front(
      new StackNode(AvailableValues, AvailableLoads, AvailableCalls,
                    CurrentGeneration, DT->getRootNode(),
                    DT->getRootNode()->begin(),
                    DT->getRootNode()->end()));

  // Save the current generation.
  unsigned LiveOutGeneration = CurrentGeneration;

  // Process the stack.
  while (!nodesToProcess.empty()) {
    // Grab the first item off the stack. Set the current generation, remove
    // the node from the stack, and process it.
    StackNode *NodeToProcess = nodesToProcess.front();

    // Initialize class members.
```

需要用到的分析结果

CSE

处理流程

```
while (!nodesToProcess.empty()) {
  // Grab the first item off the stack. Set the current generation, remove
  // the node from the stack, and process it.
  StackNode *NodeToProcess = nodesToProcess.front();

  // Initialize class members.
  CurrentGeneration = NodeToProcess->currentGeneration();

  // Check if the node needs to be processed.
  if (!NodeToProcess->isProcessed()) {
    // Process the node.
    Changed |= processNode(NodeToProcess->node());
    NodeToProcess->childGeneration(CurrentGeneration);
    NodeToProcess->process();
  } else if (NodeToProcess->childIter() != NodeToProcess->end()) {
    // Push the next child onto the stack.
    DomTreeNode *child = NodeToProcess->nextChild();
    nodesToProcess.push_front(
        new StackNode(AvailableValues,
                      AvailableLoads,
                      AvailableCalls,
                      NodeToProcess->childGeneration(), child,
                      child->begin(), child->end()));
  } else {
    // It has been processed, and there are no more children to process,
    // so delete it and pop it off the stack.
    delete NodeToProcess;
    nodesToProcess.pop_front();
  }
} // while (!nodes...)

// Reset the current generation.
CurrentGeneration = LiveOutGeneration;

return Changed;
}
```

CSE

```
bool EarlyCSE::processNode(DomTreeNode *Node) {
  BasicBlock *BB = Node->getBlock();

  // If this block has a single predecessor, then the predecessor is the parent
  // of the domtree node and all of the live out memory values are still current
  // in this block.  If this block has multiple predecessors, then they could
  // have invalidated the live-out memory values of our parent value.  For now,
  // just be conservative and invalidate memory if this block has multiple
  // predecessors.
  if (BB->getSinglePredecessor() == 0)
    ++CurrentGeneration;

  /// LastStore - Keep track of the last non-volatile store that we saw... for
  /// as long as there in no instruction that reads memory.  If we see a store
  /// to the same location, we delete the dead store.  This zaps trivial dead
  /// stores which can occur in bitfield code among other things.
  StoreInst *LastStore = 0;

  bool Changed = false;

  // See if any instructions in the block can be eliminated.  If so, do it.  If
  // not, add them to AvailableValues.
  for (BasicBlock::iterator I = BB->begin(), E = BB->end(); I != E; ) {
    Instruction *Inst = I++;

    // Dead instructions should just be removed.
    if (isInstructionTriviallyDead(Inst, TLI)) {
      DEBUG(dbgs() << "EarlyCSE DCE: " << *Inst << '\n');
      Inst->eraseFromParent();
      Changed = true;
      ++NumSimplify;
      continue;
    }
```

Writing An LLVM Pass

- 学习如何为LLVM新增Pass
- llvm/docs/WritingAnLLVMPass.rst

```
All LLVM passes are subclasses of the `Pass
<http://llvm.org/doxygen/classllvm_1_1Pass.html>`_ class, which implement
functionality by overriding virtual methods inherited from ``Pass``.  Depending
on how your pass works, you should inherit from the :ref:`ModulePass
<writing-an-llvm-pass-ModulePass>` , :ref:`CallGraphSCCPass
<writing-an-llvm-pass-CallGraphSCCPass>`, :ref:`FunctionPass
<writing-an-llvm-pass-FunctionPass>` , or :ref:`LoopPass
<writing-an-llvm-pass-LoopPass>`, or :ref:`RegionPass
<writing-an-llvm-pass-RegionPass>`, or :ref:`BasicBlockPass
<writing-an-llvm-pass-BasicBlockPass>` classes, which gives the system more
information about what your pass does, and how it can be combined with other
passes.  One of the main features of the LLVM Pass Framework is that it
schedules passes to run in an efficient way based on the constraints that your
pass meets (which are indicated by which class they derive from).
```

Writing An LLVM Pass

- 例子见lib/Transforms/Hello/
- 调用
- ./build/bin/opt -load ./build/lib/LLVMHello.so – hello ../example.s

```
1 add_llvm_loadable_module( LLVMHello
2   Hello.cpp
3   )
```

```
184 .. code-block:: c++
185
186     #include "llvm/Pass.h"
187     #include "llvm/Function.h"
188     #include "llvm/Support/raw_ostream.h"
189
190     using namespace llvm;
191
192     namespace {
193       struct Hello : public FunctionPass {
194         static char ID;
195         Hello() : FunctionPass(ID) {}
196
197         virtual bool runOnFunction(Function &F) {
198           errs() << "Hello: ";
199           errs().write_escaped(F.getName()) << '\n';
200           return false;
201         }
202       };
203     }
204
205     char Hello::ID = 0;
206     static RegisterPass<Hello> X("hello", "Hello World Pass", false, false);
207
```

提示

- DCE： dead code elimination　DCE.cpp
- LICM： loop invariant code motion　LICM.cpp
- ……

提示

- 创建自己的Pass
- 实现优化逻辑:
  - 常数传播
  - 循环不变量外提
  - 公共子表达式删除
  - 死代码删除
  - ……（自选）
- 可以模仿llvm自带的常数传播，死代码删除等进行coding
- 注意：分支、循环等情况
- 能够生成优化后的IR，和对应的可执行文件