编译原理

8. Basic Blocks and Traces

rainoftime.github.io
浙江大学
计算机科学与技术学院

课程内容

1. Introduction
2. Lexical Analysis
3. Parsing
4. Abstract Syntax
5. Semantic Analysis
6. Activation Record
7. Translating into Intermediate Code
8. **Basic Blocks and Traces**
9. Instruction Selection
10. Liveness Analysis
11. Register Allocation
13. Garbage Collection
14. Object-oriented Languages
18. Loop Optimizations

Outline

1. Canonical Form
2. Step I: Canonical Trees
3. Step II & III: Taming Conditional Branches

1. Canonical Form

Motivation

- The trees generated by semantic analysis phase must be translated into assembly or machine language
- The operators of the *Tree* language are chosen carefully to match the capabilities of most machines
- However,
  - Some aspects of the *Tree* language do not correspond exactly with machine languages
  - Some aspects of the *Tree* language interfere with compile-time optimization analyses

Mismatches: Trees vs. Machine Code

1. CJUMP can jump to two labels

- Real machines' conditional jump instructions fall through to the next instruction if the condition is false (e.g., JZ, JNZ)

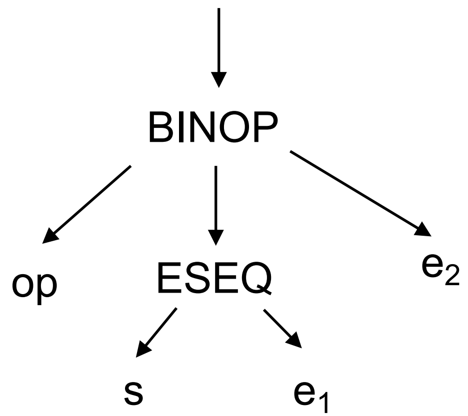
```
CJUMP(e, t, f)
...
LABEL(t)
if-true code
LABEL(f)
```

```
    evaluate e
    JZ  f
    if-true code
f:
```

- 真正的汇编指令里有conditional jump, 在条件成立会跳转; 条件不成立的情况下就**执行自己的后一条指令**
- 而在IR tree里无论成立还是不成立，都需要跳转

Mismatches: Trees vs. Machine Code

1. **CJUMP can jump to two labels (but machine code "falls through")**
2. **ESEQ nodes within expressions are inconvenient**
   – Different orders of evaluating subtrees yield different results.
   – But it is useful to be able to evaluate the subexpressions of an expression in any order.



如果计算s的时候有side-effect，那就会导致谁先做谁后做结果不一样(为什么)

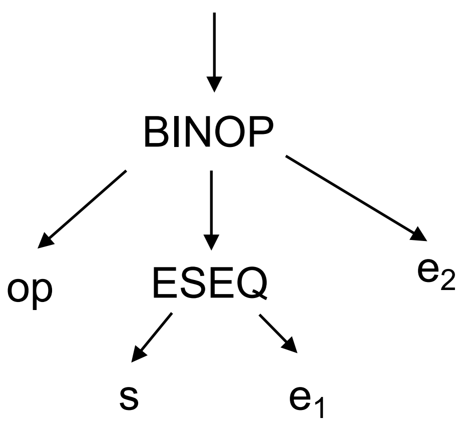
Evaluate e2 first?

回顾: 关于ESEQ的理解

- **ESEQ(s, e)**：The statement s is evaluated for side effects, then e is evaluated for a result.
  - 假设s是statement a=5, e是expression a+5
  - Statement (如a=5)不返回值,但是有副作用
  - ESEQ(a=5, a + 5)最终的结果是10
- 关于副作用(Side effects) (重要)
  - **Side-effects** means **updating** the contents of a **memory cell** or a **temporary register**

Mismatches: Trees vs. Machine Code

1. **CJUMP can jump to two labels (but machine code "falls through")**
2. **ESEQ nodes within expressions are inconvenient**
   - Different orders of evaluating subtrees yield different results.
   - But it is useful to be able to evaluate the subexpressions of an expression in any order.



如果计算s的时候有side-effect，那就会导致谁先做谁后做结果不一样

- 考虑**BINOP(PLUS, TEMP a,**
  ESEQ(MOVE(TEMP a, u), v))
- **MOVE**有副作用: 修改了临时变量/虚拟寄存器a的值！
- 也就是说, ESEQ(s, e1)可能改变了e2!

Evaluate e2 first or not?

Mismatches: Trees vs. Machine Code (Cont.)

1. **CJUMP can jump to two labels (but machine code "falls through")**
2. **ESEQ nodes within expressions are inconvenient**
3. **CALL nodes within expressions also depend on order (have side effects)**
   - When trying to put arguments into a fixed set of formal-parameter registers
   - e.g., CALL(f, [e1, CALL(g, [e2, …])])

Idea: Transform the **IR to a canonical form** to eliminate the above cases!

Why Canonical Form?

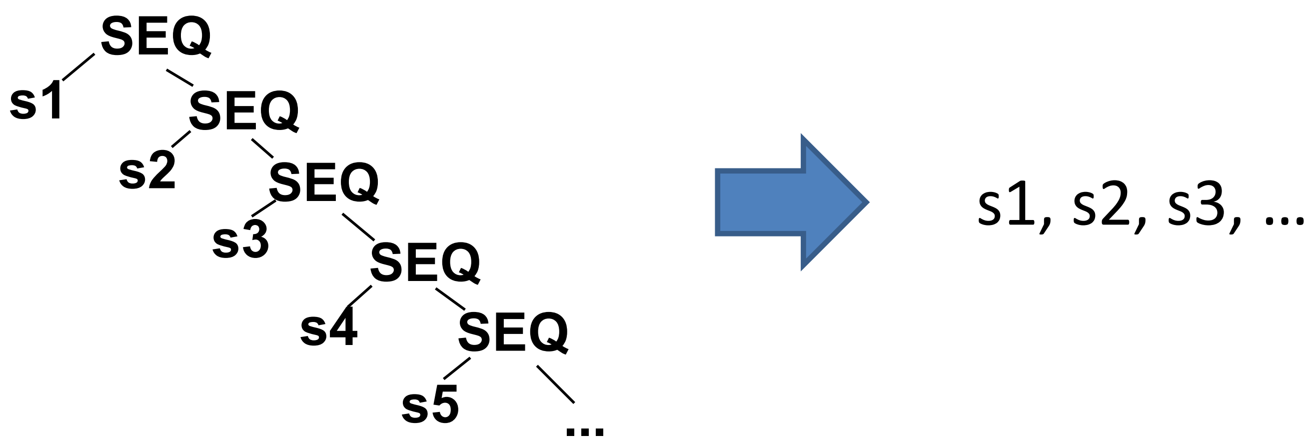
- Intermediate code has general tree form
  - Easy to generate from AST,
  - But hard to translate directly to assembly

- Assembly code is **a sequence of statements!**

- Characteristics of **canonical form**
  - All statements brought up to top level of tree
  - Can generate assembly directly

Example: Canonical Form

- In canonical form, all SEQ nodes go down right chain:



- A function is just one big SEQ containing all statements: SEQ(s1,s2,s3,s4,s5,…)
- Can translate to assembly more directly!

Transforming to Canonical Form

- To make instruction selection easier, we transform the IR tree in three stages:

  1. A tree is rewritten into a list of canonical trees without SEQ or ESEQ nodes

  2. This list is grouped into a set of basic blocks, which contain no internal jumps or labels

  3. The basic blocks are ordered into a set of traces in which every CJUMP is immediately followed by its false label

2. To Canonical Trees (Linerization)

- **eliminate ESEQs**
- move CALLs to top level
- eliminate SEQs

What are Canonical Trees

- **Canonical Trees** are defined as having these properties:
  1. **No SEQ or ESEQ**
  2. **The parent of each CALL is either EXP(…) or MOVE(TEMP t, …)**

- **Property 1:**
  – Each canonical tree only contains one statement node, i.e., the root node. Other nodes are all expression nodes.

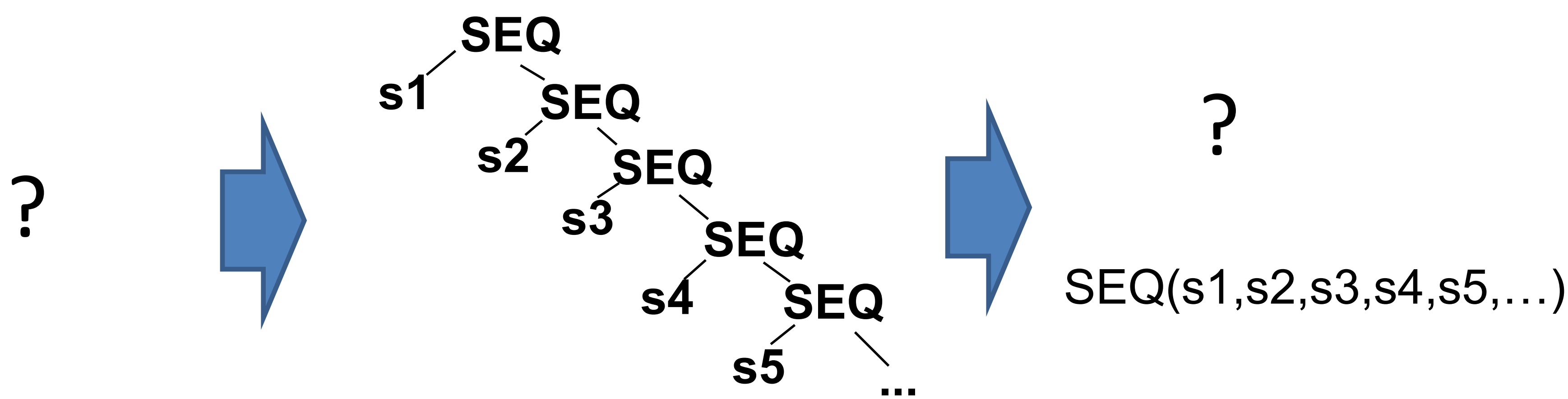
Canonical Trees

- **Canonical Trees** are defined as having these properties:
  1. **No SEQ or ESEQ**
  2. **The parent of each CALL is either EXP(…) or MOVE(TEMP t, …)**

- **Property 1:**
  - Each canonical tree only contains one statement node, i.e., the root node. Other nodes are all expression nodes.
- **Property 1 and property 2:**
  - The parent of a CALL node must be the root node of a canonical tree and must be EXP(..) or MOVE(TEMP t, ..).
  - There can only be one CALL node in a canonical tree, because EXP(…) and MOVE(TEMP t, ...) can only contains one CALL.

Stage I: To *Canonical Trees*

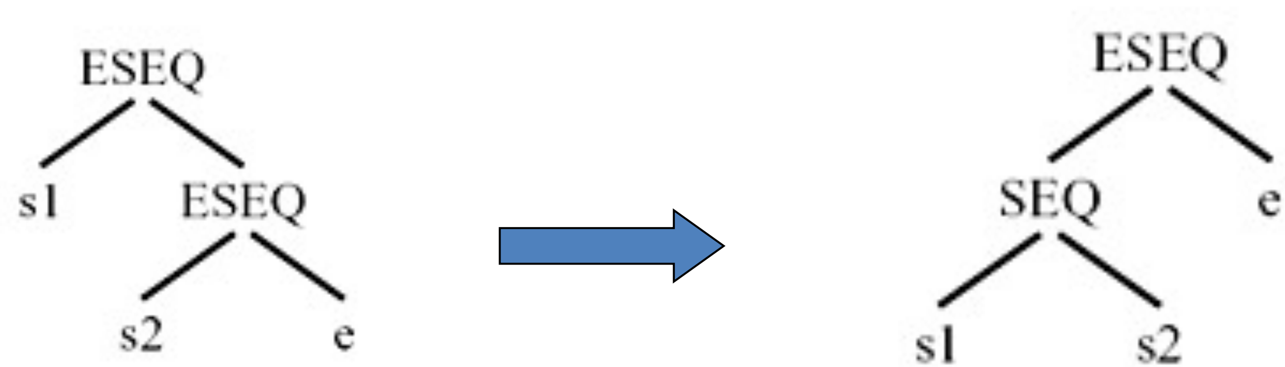
- To perform stage-one transformation, we need to:
  1. **eliminate ESEQ**
  2. **move CALLs to top level**
  3. **eliminate SEQs (turn into linear lists)**



把statements全部移上去，最后就会
和汇编比较接近(statement序列!)

Linearization Rules for ESEQ

- How can the ESEQ nodes be eliminated?
  - **Lift them higher and higher in the tree, until they can become SEQ nodes.**

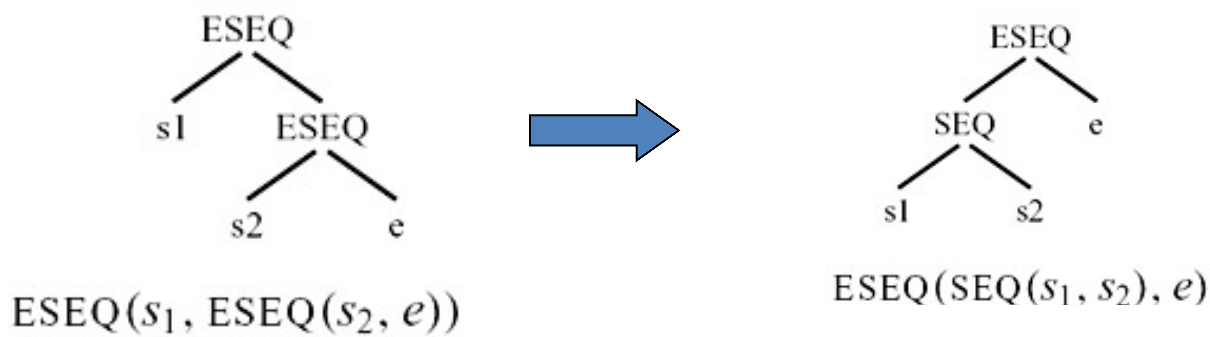


$\mathrm{ESEQ}(s_1, \mathrm{ESEQ}(s_2, e))$ $\qquad$ $\mathrm{ESEQ}(\mathrm{SEQ}(s_1, s_2), e)$

ESEQ(s1, ESEQ(s2,e)) => ESEQ(SEQ(s1,s2),e)

Linearization Rules for ESEQ Cont.

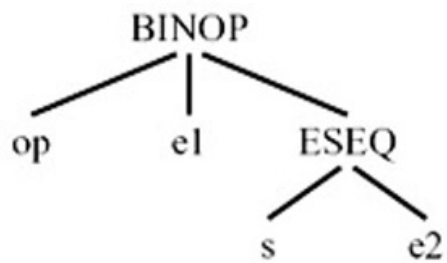
- **ESEQ(s1, ESEQ(s2, e))**$\Rightarrow$ ESEQ(SEQ(s1,s2), e))



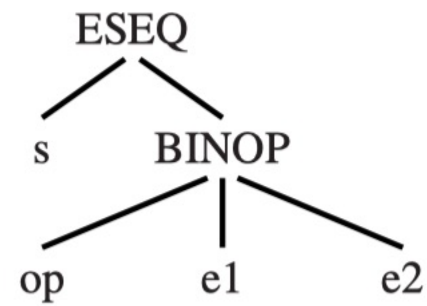
- BINOP(op, ESEQ(S, e1,), e2) $\Rightarrow$ ESEQ(s, BINOP(op, e1, e2))
- MEM(ESEQ(s,e1)) $\Rightarrow$ ESEQ(s, MEM(e1))
- JUMP(ESEQ(s, e1)) $\Rightarrow$ SEQ(s, JUMP(e1))
- CJUMP(op, ESEQ(s, e1), e2, l1,l2) $\Rightarrow$ SEQ(s, CJUMP(op, e1, e2, l1,l2))

Impact of Side Effects on Linearization Rules

Consider the Tree: BINOP(op, e1, ESEQ(s, e2))



Can we ?



$\mathrm{BINOP}(op, e_1, \mathrm{ESEQ}(s, e_2))$ $\qquad$ $\mathrm{ESEQ}(s, \mathrm{BINOP}(op, e_1, e_2))$

Can we interchange the order of s and e1??

- **Problem**: s may have side effects that affect value of e1!
- **Solution**: use a temporary to store value of e1

Suppose:
s = MOVE(MEM(x), y)
e1 =MEM(x)

Side Effects and Commutativity

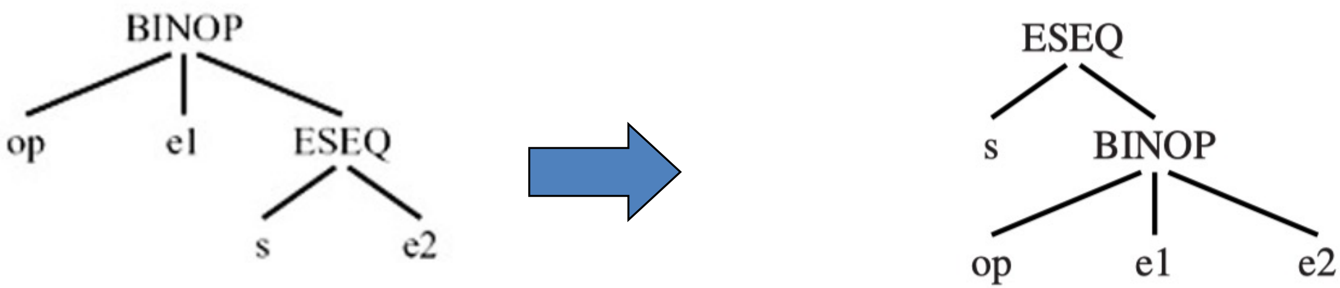
- **Commutativity**: statement **s** and expression **e** can commute if **s** does not affect the value of **e**
  - E.g., consider MOVE (MEM(t1), e) and MEM(t2)
  - If t1 != t2, then MOVE (MEM(t1), e) and MEM(t2) commute

- What if statement **s** and expression **e** do not commute?
  - we may need **new temporary locations to store intermediate results** to get canonical trees

Effect of Commutativity on Linearization Rules

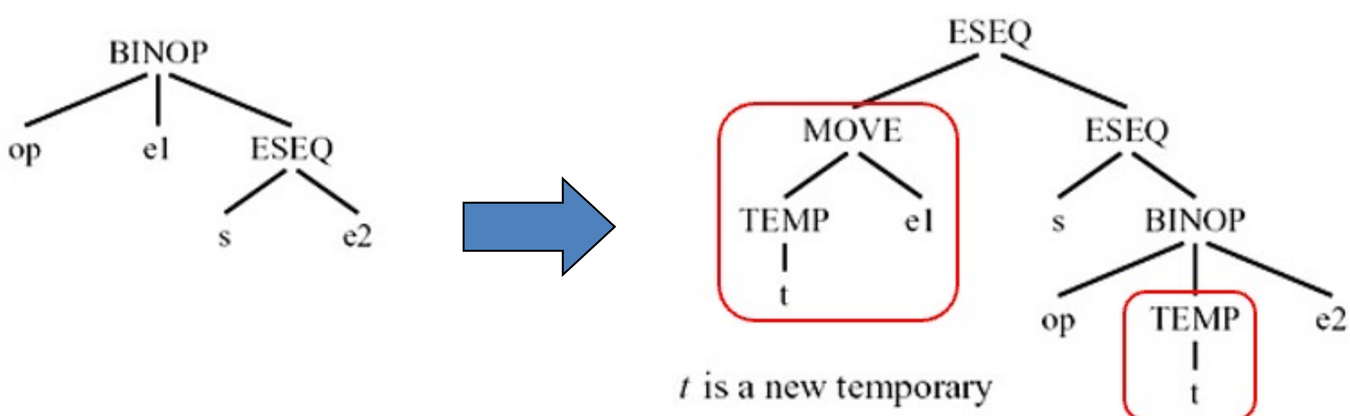
Consider BINOP(op, e1, ESEQ(s, e2))

Statement **s** and expression **e** commute



$\text{BINOP}(op, e_1, \text{ESEQ}(s, e_2))$ $\text{ESEQ}(s, \text{BINOP}(op, e_1, e_2))$

Statement **s** and expression **e** do not commute



t is a new temporary

$\text{BINOP}(op, e_1, \text{ESEQ}(s, e_2))$

$\text{ESEQ}(\text{MOVE}(\text{TEMP}\ t, e_1),$
$\quad \text{ESEQ}(s, \text{BINOP}(op, \text{TEMP}\ t, e_2)))$

Example: Effect of Commutativity on Linearization

If s, e1 commute, we have

- **BINOP(op, e1, ESEQ(s, e2))** $\Rightarrow$ ESEQ(s, BINOP(op, e1, e2))
- **CJUMP(op, e1, ESEQ(s, e2), l1,l2)** $\Rightarrow$ SEQ(s, CJUMP(op, e1, e2, l1,l2)

Else, we use the following rules (that have new temporaries)

- **BINOP(op, e1, ESEQ(s, e2))**

$\Rightarrow$ ESEQ(MOVE(TEMP t, e1),
ESEQ(s, BINOP(op, TEMP t, e2)))

- **CJUMP(op, e1, ESEQ(s, e2), l1, l2)**

$\Rightarrow$ SEQ(MOVE(TEMP t, e1),
SEQ(s, CJUMP(op, TEMP t, e2, l1,l2)))

Deciding Commutativity

Whether a statement s commutes with an expression e?

- **Problem**: commutativity is hard to known statically.
- Make a **conservative approximation**, which means
  - commute(s, e) = True if s and e definitely do commute
  - commute(s, e) = False otherwise
- E.g., a naïve strategy to estimate whether a statement commutes with an expression:
  - A constant commutes with any statement
  - An empty statement commutes with any expression.
  - **Anything else is assumed not to commute**

Deciding Commutativity (此页不要求掌握)

- Rules for BINOP and MOVE rely on the interchanging the order of a lowered statement s and an expression e
  - This can be done safely when the statement cannot alter the value of the expression

- **Two conditions that s and e cannot be interchanged**
  1. The statement could **change** the value of **a temporary variable** used by the expression
  2. The statement could **change** the value of **a memory location** used by the expression.

Deciding Commutativity (此页不要求掌握)

How to check the above two conditions?

- **Temporaries**: It is easy to determine whether the statement updates a temporary used by the expression, because temporaries have unique names.
- **Memory**: It is much harder because two memory locations can be **aliases**!
  – The statement s uses a memory location MME(e1) as a destination
  – The expression reads from the memory location MEM(e2)
  – e1 might have the save value as e2!!

More precise (still conservative) approximation: use some **alias analyses**!

“Falcon: A Fused Approach to Path-sensitive Sparse Data Dependence Analysis,” PLDI 2024.

2. To Canonical Trees (Linerization)

- eliminate ESEQs
- **move CALLs to top level**
- eliminate SEQs

Moving Calls to Top Level

- How to implement CALL expression?
  - Assign the return value to a dedicated return-value register (to reduce memory traffic)
  - For example, eax/rax on x86/x86-64

- Consider BINOP(PLUS, CALL(...), CALL(...))
  - The second call will overwrite the RV register before the PLUS can be execute

像这样的式子，树结构逻辑上也没问题，但机器实现就出问题。因为函数的返回值都是用同一个寄存器（rax）来存的，连续运算两个，有一个就丢失了，然后再op就没法做。

Move CALLS to Top Level

- **Idea**: assign each return value immediately into a fresh temporary register:

```
CALL(fun, args) ->
    ESEQ(MOVE(TEMP t, CALL(fun, args)), TEMP t)
```

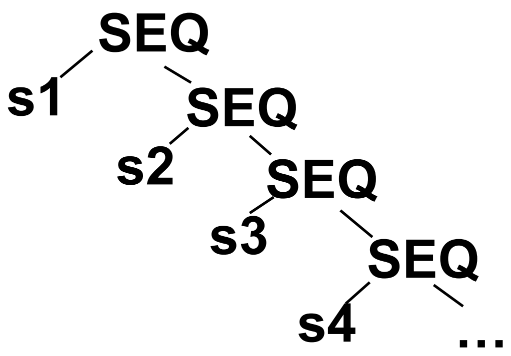
2. To Canonical Trees (Linerization)

- eliminate ESEQs
- move CALLs to top level
- **eliminate SEQs**

A Linear List of Statements

- After applying the above rules, we obtain

SEQ(SEQ(SEQ(..., sx), sy), sz)



- Next, we repeatedly apply the rule:

SEQ(SEQ(a, b), c) = SEQ(a, seq(b, c))

- And obtain a statement of the form

SEQ(s_1, SEQ(s_2, ..., SEQ(s_{n-1},s_n)...))

- We can just consider this to be a simple list of statements:

s_1, s_2, ..., s_{n-1}, s_n

(None of the s_i contain SEQ/ESEQ nodes)

3. Taming Conditional Branches

- **Basic Blocks**
- **Traces**

Taming Conditional Branches

- **Problem of CJUMP**: NO counterpart for two-way branch on most machines
- **Goal**: rearrange the trees so that CJUMP(cond, l_t, l_f) is immediately followed by LABEL(l_f)

```
CJUMP xxx α β
```

```
CJUMP xxx Label_alpha
β ...
Label_alpha:
α
```

- **Solution**: two-stages approach
  1. Form a list of canonical trees into basic blocks
  2. Order the basic blocks into traces

Solution

- We transform the tree in three stages:

1. ~~A tree is rewritten into a list of *canonical trees* without SEQ or ESEQ nodes~~
2. **This list is grouped into a set of basic blocks, which contain no internal jumps or labels**
3. The basic blocks are ordered into a set of traces in which every CJUMP is immediately followed by its false label.

Basic Blocks

$\mathsf{LABEL}(l)$
...
$\mathsf{CJUMP}(e,\ l_1,\ l_2)$

- A **basic block** is
  - A sequence of statements that is always entered at the beginning and exited at the end

- That is:
  1. The first statement is a LABEL
  2. The last statement is a JUMP or CJUMP
  3. There are no other LABELs, JUMPs, or CJUMPs I the block

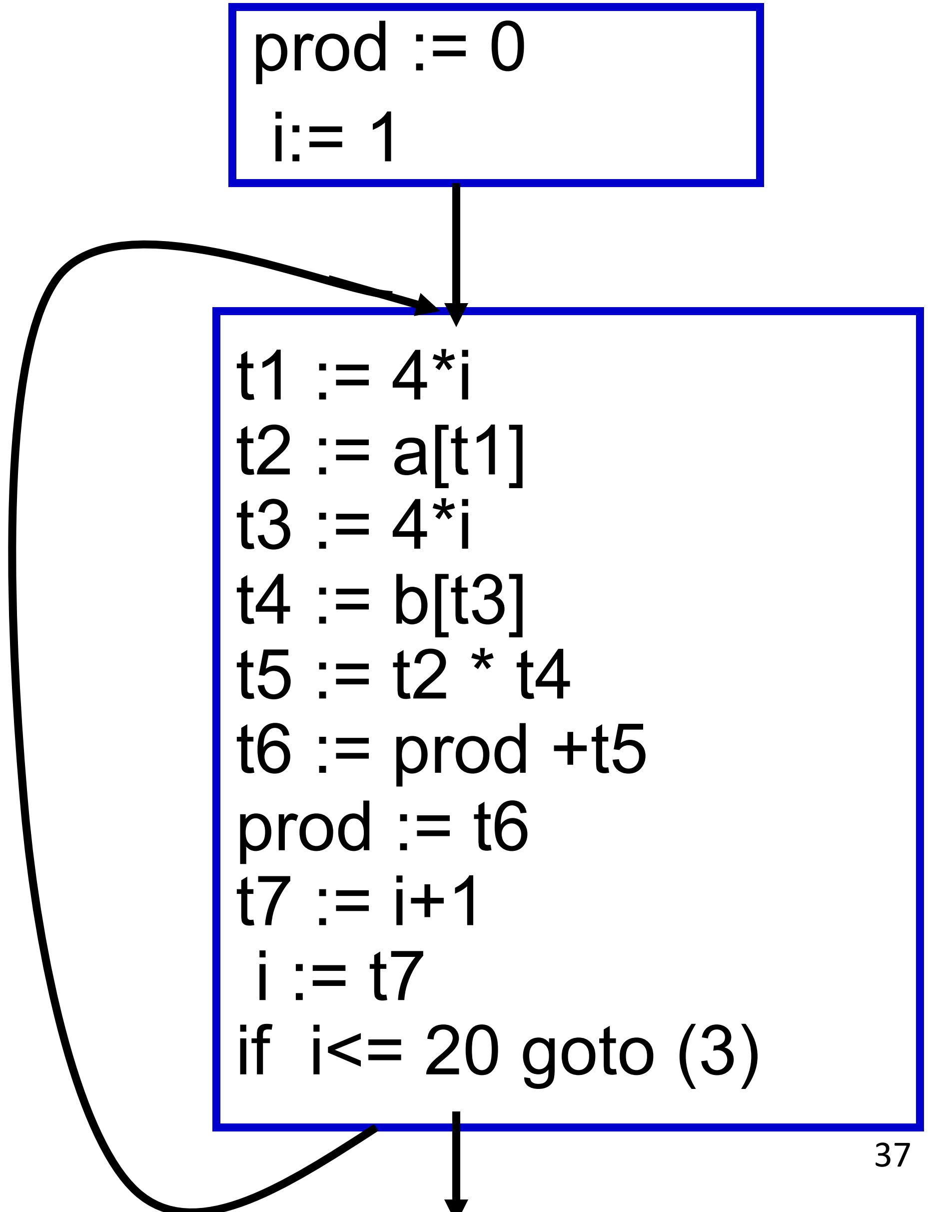
Algorithm for Basic Block Construction

Given sequence of intermediate code statements

1. Scan from beginning to end
2. When a label is found, start a new block (and end the previous block)
3. Whenever a cjump/jump is found the current block is ended (and the next block is started)
4. If this leaves a block ending without a cjump/jump, then append a jump to the next block
5. If a block has no label at the begining, invent one, and add it

Example: Basic Blocks

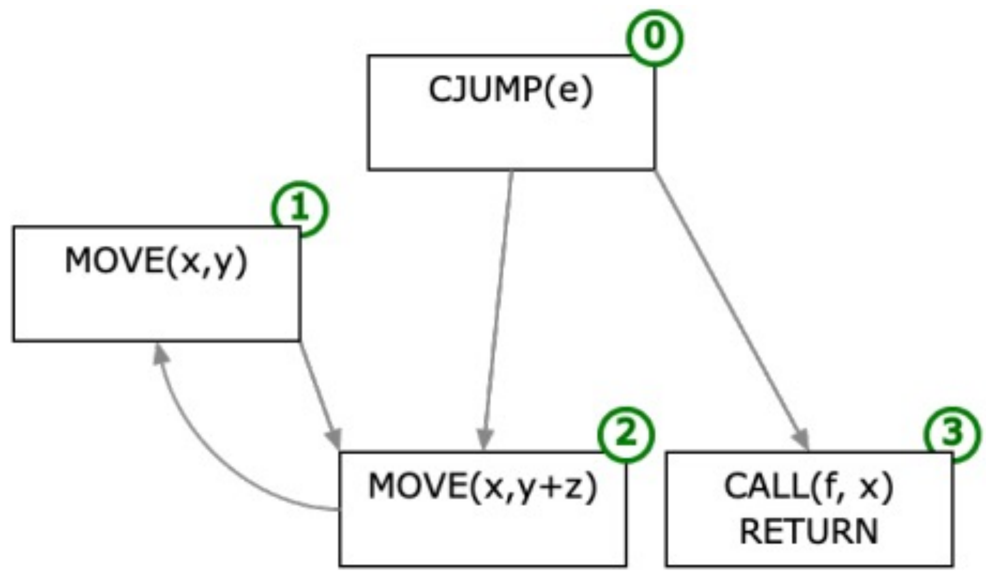
```
(1)  prod := 0
(2)  i:= 1
(3)  t1 := 4*i
(4)  t2 := a[t1]
(5)  t3 := 4*i
(6)  t4 := b[t3]
(7)  t5 := t2 * t4
(8)  t6 := prod +t5
(9)  prod := t6
(10) t7 := i+1
(11) i := t7
(12) if  i<= 20 goto (3)
```



Example: Basic Blocks

- Control flow graph (CFG): nodes are basic blocks and edges are jumps between them
  - In some contexts, the node of a CFG is a statement (to be discussed in the register allocation section)

L0:	CJUMP(e, L2, L3)
L1:	MOVE(x, y)
L2:	MOVE(x, y + z) JUMP(L1)
L3:	CALL(f, x) RETURN



3.Taming Conditional Branches

- Basic Blocks
- Traces

Solution

- How to eliminate these mismatches?
- We transform the tree in three stages:

1. ~~A tree is rewritten into a list of *canonical trees* without SEQ or ESEQ nodes~~
2. ~~This list is grouped into a set of basic blocks, which contain no internal jumps or labels~~
3. **The basic blocks are ordered into a set of traces, in which every CJUMP is immediately followed by its false label.**

Recap: Taming Conditional Branches

- **Issue of CJUMP**: NO counterpart for two-way branch on most machines
- **Problem Statement**: rearrange the trees so that CJUMP(cond, l_t, l_f) is immediately followed by LABEL(l_f)

```
CJUMP xxx α β
```

```
CJUMP xxx Label_alpha
β ...
Label_alpha:
α
```

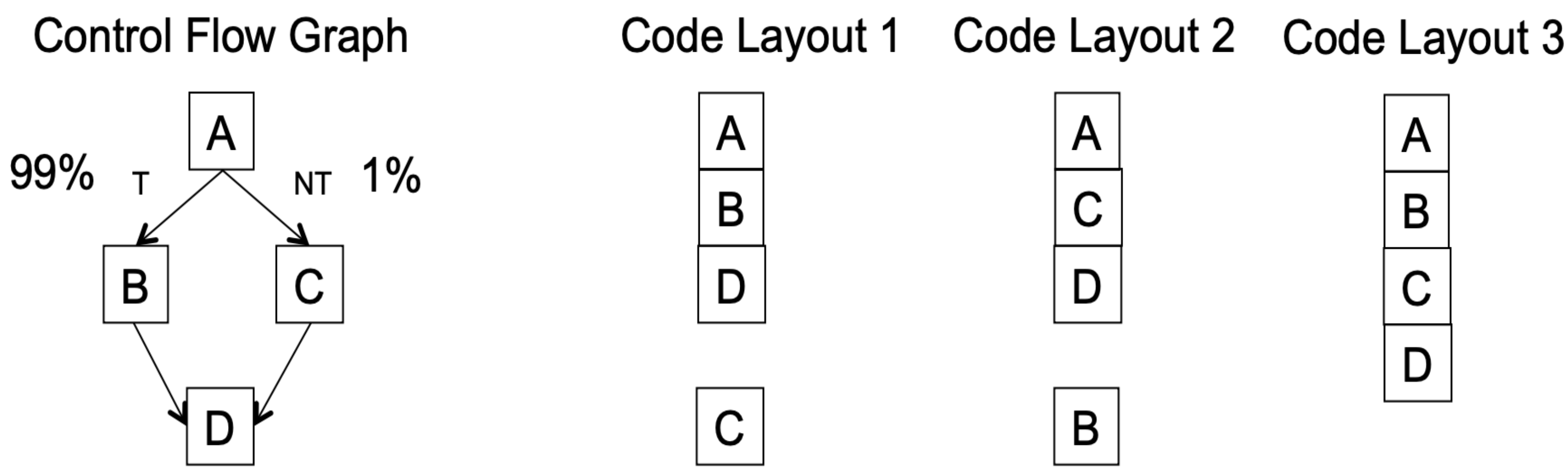
- **Solution**: two-stages approach
  1. Form a list of canonical trees into basic blocks
  2. **Reorder the basic blocks into traces**

Basic Block Reordering

- The basic blocks can be arranged in any order, and the result of executing the program will be the same
- Based on this property, we can optimize the ***nature*** and ***number*** of jumps:
  1. Choose an ordering of the blocks such that **each CJUMP is followed by its false label**
  2. Arrange that many of the unconditional JUMPs are immediately followed by their target label
     - Allow the **deletion of the unconditional jumps**, making the compiled program run a bit faster.
  3. (Other aspects: may also optimize instruction cache, etc.)

Example: Basic Block Reordering

- How to generate target code from CFG?
- The basic blocks can be arranged in any order, and the result of executing the program will be the same



- 90%, 10%: the execution frequency (from dynamic profiling)
- Code Layout 1 reduces fetch breaks, increases I-cache hit rate,…

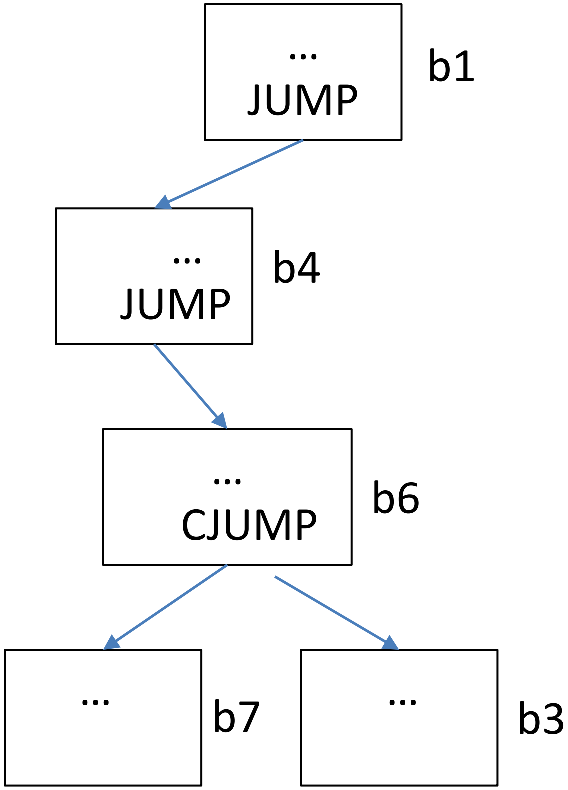
“Profile Guided Code Positioning,” PLDI 1990.

Traces for Basic Block Reording

- The usual technique for finding a good ordering of basic blocks is to construct traces
- **Trace:** A sequence of statements that could be consecutively executed during the execution
    - Alternatively, a sequence of basic blocks
- **A covering set of traces**
  - Each trace is loop free
  - Each block must be in exactly one trace

Basic Rule for Generating ONE Trace

- Suppose block $b1$ ends with a **JUMP** to $b4$, and $b4$ has a JUMP to $b6$. Then, we can make the trace ***b*1, *b*4, *b*6**.
- Suppose b6 ends with a conditional jump **CJUMP**(cond, b7, b3). We append b3 to our trace and continue with the rest of the trace after b3.
- The basic block b7 will be in some other trace.



Make sure that CJUMP(cond, l_t, l_f) is immediately followed by LABEL(l_f)!

Generating a Covering Set of Traces

Put all the blocks of the program into a **list Q**
While **Q** is not empty
    start a **new (empty) trace**, call it **T**
    remove the **head element** **b** from **Q**
    while **b** is not marked
        mark **b**; append **b** to the end of the current trace **T**;
        examine the **successors** of **b**
        if there is any unmarked successor c
            b $\leftarrow$ c
    end the current trace T

Algorithm 8.3 or Tiger Book

- Start with some block and follow a chain of jumps, marking each block and appending it to the **current trace**
- When coming to a block whose successors are all marked, the generation of **one trace is finished** → Pick an unmarked block to start the **next trace**

Generating a Covering Set of Traces

迭代式计算covering sets of traces

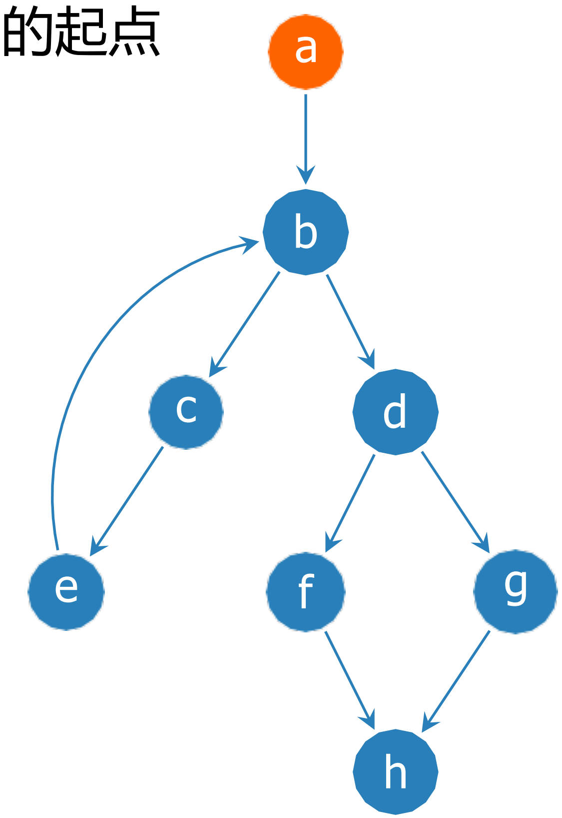
- **如何计算1个trace**
  - 从某个basic block开始，往后继节点遍历，标记每个被访问的basic block并将其附加到当前trace中
  - 当到达某basic block后继节点均已标记, 这个trace就算完了
- **如何计算新的trace**
  - 选择一个未标记的basic block作为下一个trace的起点
- **全局终止条件**
  - 不断迭代、直到所有的basic blocks都被标记了

Example: Generating a Covering Set of Traces

- **Basic algorithm**: depth-first traversal of the CFG
  - 从某个basic block开始，往后继节点遍历，标记每个被访问的basic block并将其附加到当前trace中
  - 当到达某个basic block, 其后继节点均已标记, 这个trace就算完了
  - 选择一个未标记的basic block作为下一个trace的起点

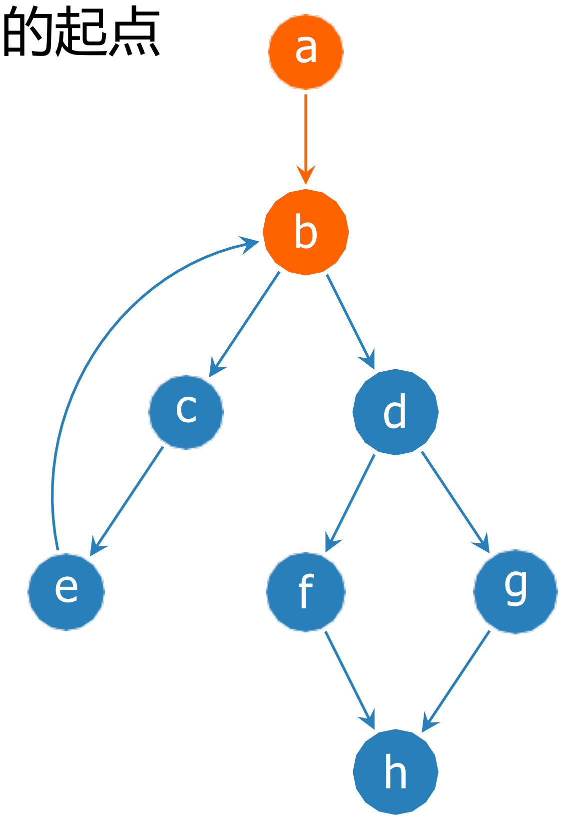
Covering set of traces

- ?



Example: Generating a Covering Set of Traces

- **Basic algorithm**: depth-first traversal of the CFG
  - 从某个basic block开始，往后继节点遍历，标记每个被访问的basic block并将其附加到当前trace中
  - 当到达某个basic block, 其后继节点均已标记, 这个trace就算完了
  - 选择一个未标记的basic block作为下一个trace的起点

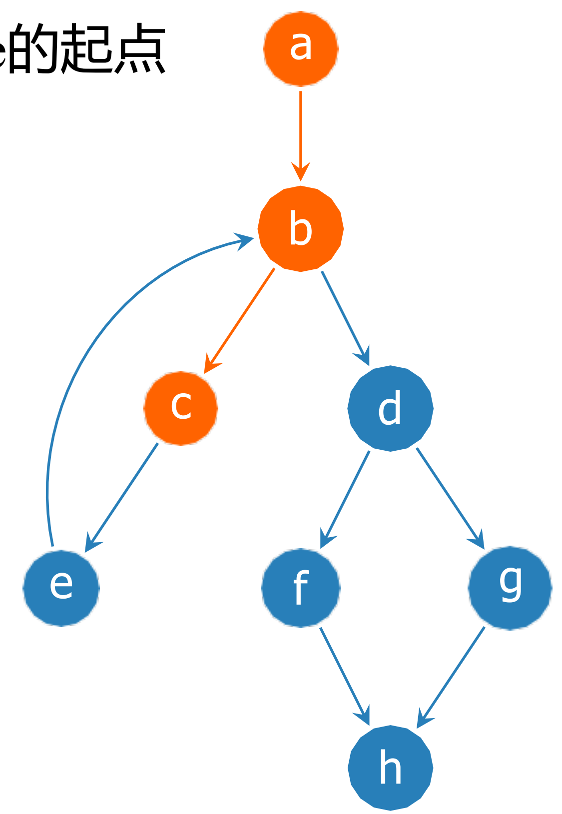


Covering set of traces

- ?

Example: Generating a Covering Set of Traces

- **Basic algorithm**: depth-first traversal of the CFG
  - 从某个basic block开始，往后继节点遍历，标记每个被访问的basic block并将其附加到当前trace中
  - 当到达某个basic block, 其后继节点均已标记, 这个trace就算完了
  - 选择一个未标记的basic block作为下一个trace的起点

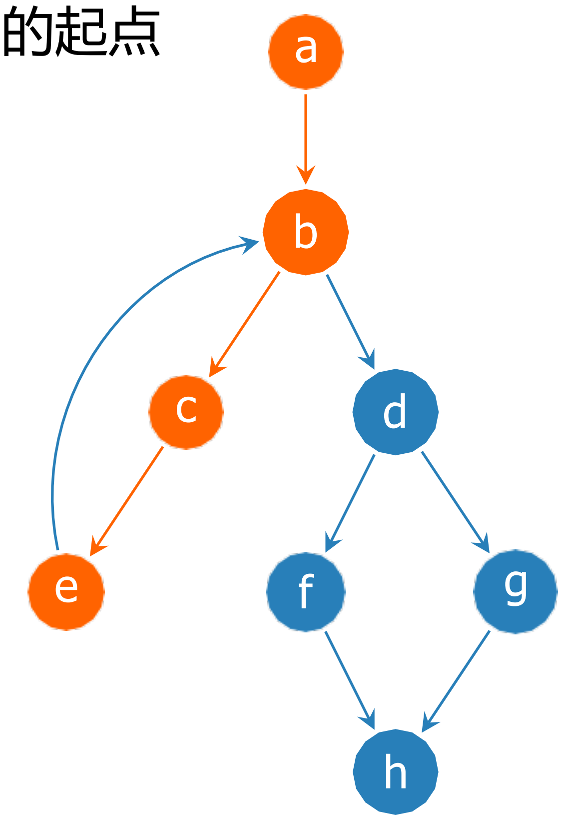


Covering set of traces

- ?

Example: Generating a Covering Set of Traces

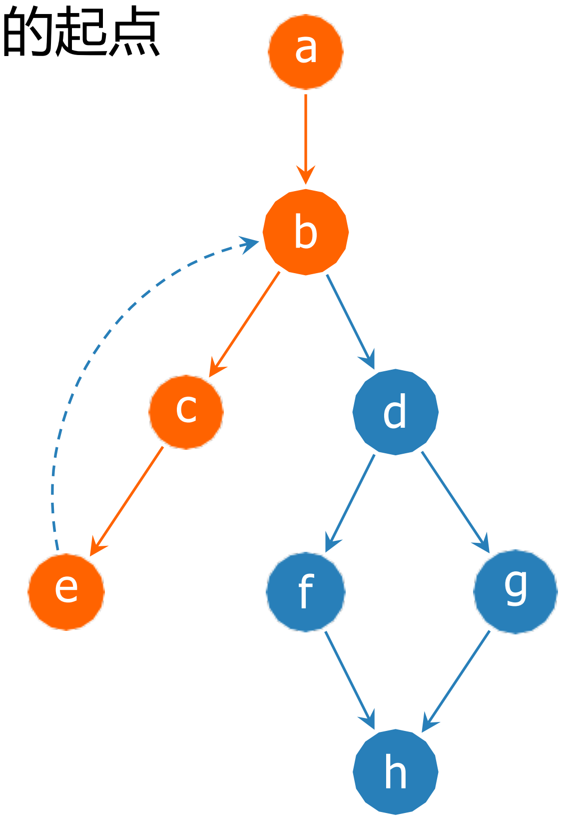
- **Basic algorithm**: depth-first traversal of the CFG
  - 从某个basic block开始，往后继节点遍历，标记每个被访问的basic block并将其附加到当前trace中
  - 当到达某个basic block, 其后继节点均已标记, 这个trace就算完了
  - 选择一个未标记的basic block作为下一个trace的起点



Covering set of traces
- ?

Example: Generating a Covering Set of Traces

- **Basic algorithm**: depth-first traversal of the CFG
  - 从某个basic block开始，往后继节点遍历，标记每个被访问的basic block并将其附加到当前trace中
  - 当到达某个basic block, 其后继节点均已标记, 这个trace就算完了
  - 选择一个未标记的basic block作为下一个trace的起点

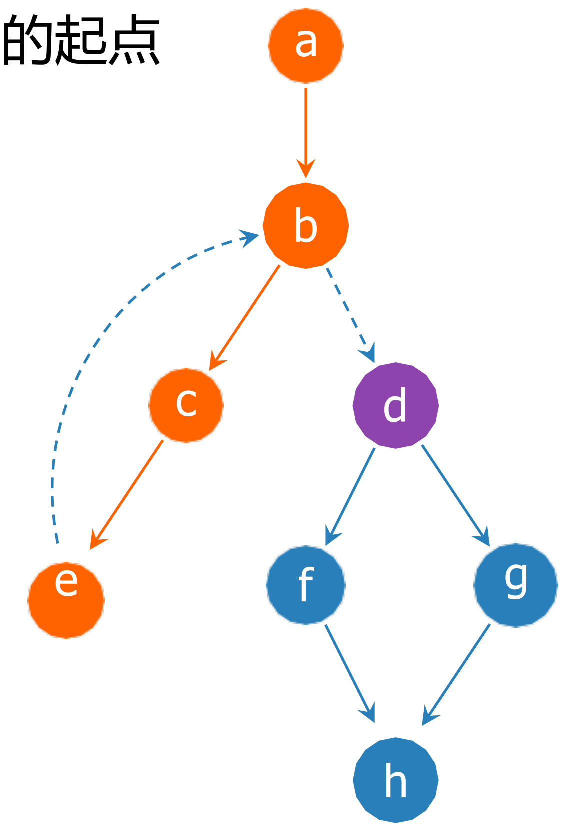


Covering set of traces

- {a, b, c, d}

Example: Generating a Covering Set of Traces

- **Basic algorithm**: depth-first traversal of the CFG
  - 从某个basic block开始，往后继节点遍历，标记每个被访问的basic block并将其附加到当前trace中
  - 当到达某个basic block, 其后继节点均已标记, 这个trace就算完了
  - 选择一个未标记的basic block作为下一个trace的起点



Covering set of traces

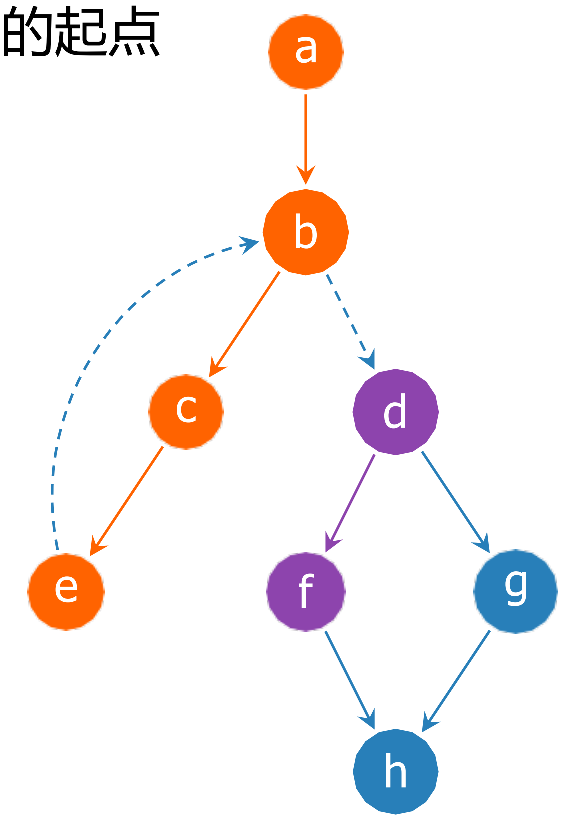
- {a, b, c, d}

Example: Generating a Covering Set of Traces

- **Basic algorithm**: depth-first traversal of the CFG
  - 从某个basic block开始，往后继节点遍历，标记每个被访问的basic block并将其附加到当前trace中
  - 当到达某个basic block, 其后继节点均已标记, 这个trace就算完了
  - 选择一个未标记的basic block作为下一个trace的起点

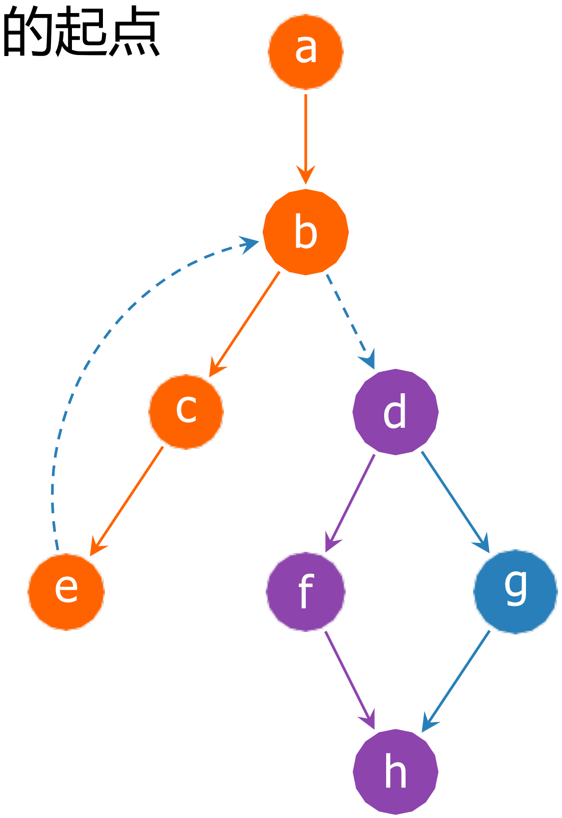
Covering set of traces

- {a, b, c, d}



Example: Generating a Covering Set of Traces

- **Basic algorithm**: depth-first traversal of the CFG
  - 从某个basic block开始，往后继节点遍历，标记每个被访问的basic block并将其附加到当前trace中
  - 当到达某个basic block, 其后继节点均已标记, 这个trace就算完了
  - 选择一个未标记的basic block作为下一个trace的起点



Covering set of traces

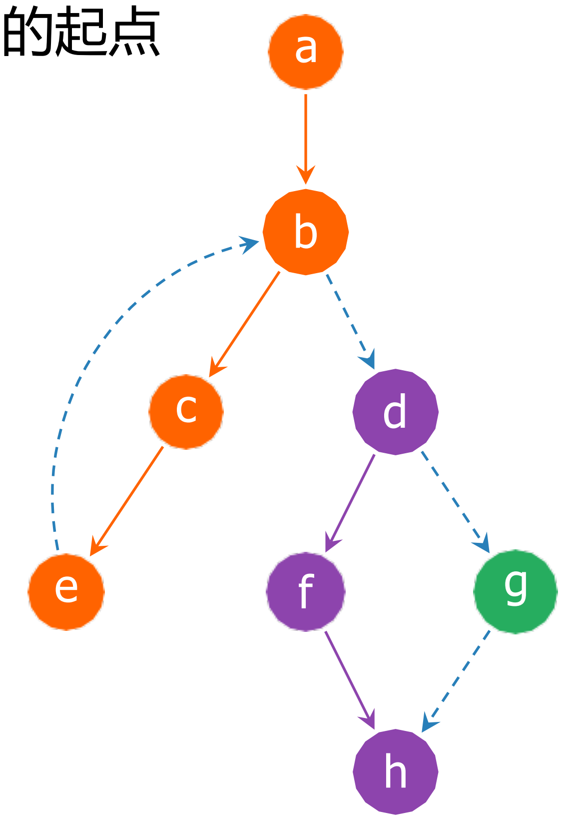
- {a, b, c, d}
- {d, f, h}

Example: Generating a Covering Set of Traces

- **Basic algorithm**: depth-first traversal of the CFG
  - 从某个basic block开始，往后继节点遍历，标记每个被访问的basic block并将其附加到当前trace中
  - 当到达某个basic block, 其后继节点均已标记, 这个trace就算完了
  - 选择一个未标记的basic block作为下一个trace的起点

Covering set of traces

- {a, b, c, d}
- {d, f, h}
- {g}



Finishing Up (JUMP Consideration)

- We prefer CJUMP followed by its false label, since this translates to machine code conditional jump
  - **For any CJUMP followed by its true label**
    - Switch the true and false labels and negate the condition.
  - **For any CJUMP immediately followed by its false label**
    - We let alone (there will be many of these).
  - **For any CJUMP(cond, a, b, lt, lf) followed by neither label, replace with**

CJUMP(*cond, a, b, lt, lf '*)
LABEL *lf '*
JUMP(NAME *lf*)

- Remove all JUMPS followed by their targe LABLES (remove unconditional jumps)

Optimal Traces

- "Optimality" needs criteria

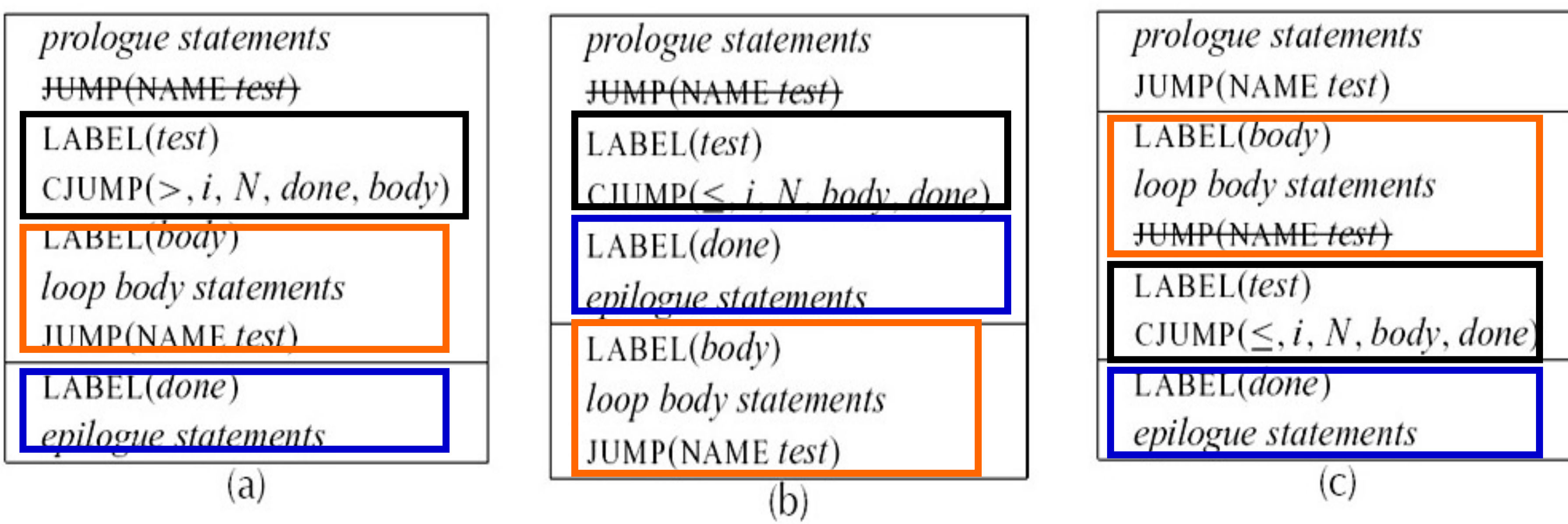
- E.g., any frequently executed sequence of instructions (such as the body of a loop) should occupy its own trace.
  - This helps to reduce the number of unconditional jumps
  - This helps with other kinds of optimizations.
    - register allocation
    - instruction scheduling
    - …

Example: Optimal Traces

(a): While循环的每个迭代有一个CJUMP和一个JUMP

(b): 使用了不同traces, 但每个迭代仍有一个CJUMP和一个JUMP

(c): 每个迭代都没有JUMP



Summary

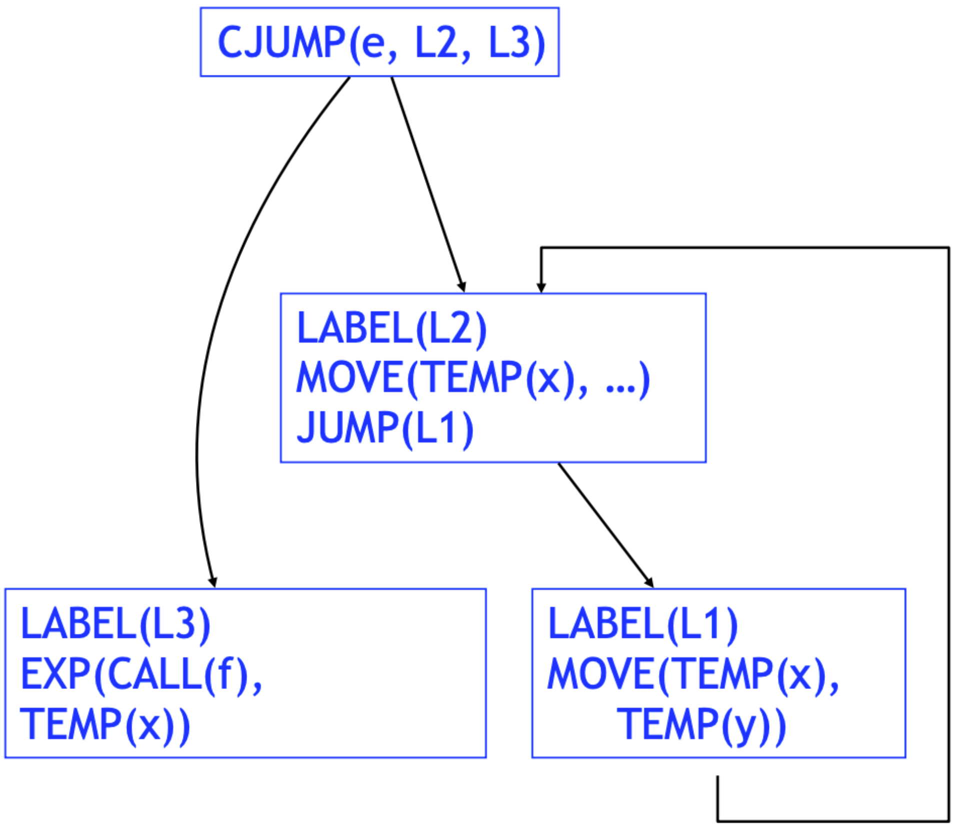
- **Problem**: mismatches between tree code and machine instructions:
  - 1. CJUMP to two labels; machine conditionals fall through on false
  - 2. ESEQ and CALL order evaluation of subtrees matters (side-effects)
  - 3. CALL as argument to another CALL causes interference between register arguments
- **Idea**: rewrite to equivalent trees without these cases
  - SEQ can only be subtree of another SEQ
  - SEQs clustered at top of tree
  - might as well turn into simple linear list of statements
- **Approach**: 3-stage transformation:
  - To linear list of canonical trees without SEQ/ESEQ
  - To basic blocks with no internal jumps or labels
  - To traces with every CJUMP immediately followed by false target

Overview of IR → Machine Code

- **Step #1** : Transform the IR trees into **a list of canonical trees**
  - a. eliminate SEQ and ESEQ nodes
  - b. the arguments of a CALL node should never be other CALL nodes
- **Step #2** : Rearrange the **canonical trees** (into **traces**) so that every CJUMP(cond,lt,lf) is immediately followed by LABEL(lf)
- **Step #3** : **Instruction Selection** --- generate the pseudo-assembly code from the canonical trees in the step #2
- **Step #4** : Perform **register allocations** on pseudo-assembly code

Thank you all for your attention

Example: Reordered Code



CJUMP(e, L2, [L3])
JUMP(L3)

LABEL(L2)
MOVE(TEMP(x), TEMP(y) +
TEMP(z))
~~JUMP(L1)~~

LABEL(L1)
MOVE(TEMP(x), TEMP(y)
JUMP(L2)

LABEL(L3)
EXP(CALL(NAME(f)), TEMP(x))

Rules for Canonical Trees Construction

ESEQ(s1, ESEQ(s2, e))	⇒	ESEQ(SEQ(s1,s2), e))
BINOP(op, ESEQ(S, e1,), e2)	⇒	ESEQ(s, BINOP(op, e1, e2))
MEM(ESEQ(s,e1))	⇒	ESEQ(s, MEM(e1))
JUMP(ESEQ(s, e1))	⇒	SEQ(s, JUMP(e1))
CJUMP(op, ESEQ(s, e1), e2, l1,l2)	⇒	SEQ(s, CJUMP(op, e1, e2, l1,l2))
BINOP(op, e1, ESEQ(s, e2))	⇒	ESEQ(MOVE(TEMP t, e1), ESEQ(s, BINOP(op, TEMP t, e2)))
CJUMP(op, e1, ESEQ(s, e2), l1, l2)	⇒	SEQ(MOVE(TEMP t, e1), SEQ(s, CJUMP(op, TEMP t, e2, l1,l2)))
MOVE(ESEQ(s, e1), e2)		SEQ(s, MOVE(e1, e2))
CALL(f , a)		ESEQ(MOVE(TEMP t, CALL(f , a)), TEMP(t))

ESEQ

- 简单说就是把ESEQ往上提，考虑ESEQ(s, e)，移动s和e表达式的时候，只要s在e前面执行，并且最后返回值是e，ESEQ的语义就保留了。
- 但问题是，移动之后，s和e之间就有了其他的表达式，这些表达式的计算如果和s有关系，那么也会被s的副作用影响，而这是不对的。
- 我们并不总是能在编译阶段判断两个表达式之间是否有影响，这意思是说，像CONST的语句，肯定无影响，但MEM就无法判断是否是同一个数据，
- 所以，对于无影响的，我们就能比较简单的交换顺序，有影响的，我们可以通过多做一次MOVE来消除影响

BINOP(op, e1, ESEQ(s, e2))=>ESEQ(MOVE(TEMP t, e1), ESEQ(s, BINOP(op, TEMP t, e2)));

也就是说，用中间变量保存可能受影响的变量