Lecture 25 **CIS 4521/5521: COMPILERS**

Announcements

- HW6: Analysis & Optimizations
 - Alias analysis, constant propagation, dead code elimination, register allocation
 - Due: Wednesday, April 30th at 10:00pm
 - Posted test case due Tuesday, April 29th at 10:00pm
- Lecture Cancelled this Thursday 4/24
 - Dr. Zdancewic will be out of town
- Final Exam:
 - According to registrar: Thursday, May 8th noon 2:00pm
 - Coverage: emphasizes material since the midterm
 - Cheat sheet: one, hand-written, double-sided, letter-sized page of notes

LOOPS AND DOMINATORS

Zdancewic CIS 4521/5521: Compilers

Loops in Control-flow Graphs

- Taking into account loops is important for optimizations.
 - The 90/10 rule applies, so optimizing loop bodies is important
- Should we apply loop optimizations at the AST level or at a lower representation?
 - Loop optimizations benefit from other IR-level optimizations and vice-versa, so it is good to interleave them.
- Loops may be hard to recognize at the quadruple / LLVM IR level.
 Many kinds of loops: while, do/while, for, continue, goto...
- Problem: *How do we identify loops in the control-flow graph?*

Definition of a Loop

- A *loop* is a set of nodes in the control flow graph.
 - One distinguished entry point called the *header*
- Every node is reachable from the header & the header is reachable from every node.
 - A loop is a strongly connected component
- No edges enter the loop except to the header
- Nodes with outgoing edges are called loop exit nodes



Nested Loops

- Control-flow graphs may contain many loops
- Loops may contain other loops:





The control tree depicts the nesting structure of the program.

Control-flow Analysis

- Goal: Identify the loops and nesting structure of the CFG.
- Control flow analysis is based on the idea of *dominators*:
- Node A *dominates* node B if the only way to reach B from the start node is through node A.
- An edge in the graph is a *back edge* if the target node dominates the source node.
- A loop contains at least one back edge.



Dominator Trees

- Domination is transitive:
 - if A dominates B and B dominates C then A dominates C
- Domination is anti-symmetric:
 - if A dominates B and B dominates A then A = B
- Every flow graph has a dominator tree
 - The Hasse diagram of the dominates relation



Dominator Dataflow Analysis

- We can define Dom[n] as a forward dataflow analysis.
 - Using the framework we saw earlier: Dom[n] = out[n] where:
- "A node B is dominated by another node A if A dominates *all* of the predecessors of B."
 - in[n] := $\bigcap_{n' \in pred[n]} out[n']$
- "Every node dominates itself."

 $- \text{ out}[n] := in[n] \cup \{n\}$

- Formally: $\mathcal{L} = \text{set of nodes ordered by } \subseteq$
 - $T = \{all nodes\}$
 - $\ \ F_n(x) = x \ U \ \{n\}$
 - ∏ is ∩
- Easy to show monotonicity and that F_n distributes over meet.
 - So algorithm terminates and is MOP

Improving the Algorithm

- Dom[b] contains just those nodes along the path in the dominator tree from the root to b:
 - e.g., $Dom[8] = \{1, 2, 4, 8\}, Dom[7] = \{1, 2, 4, 5, 7\}$
 - There is a lot of sharing among the nodes
- More efficient way to represent Dom sets is to store the dominator *tree*.
 - doms[b] = immediate dominator of b
 - doms[8] = 4, doms[7] = 5
- To compute Dom[b] walk through doms[b]
- Need to efficiently compute intersections of Dom[a] and Dom[b]
 - Traverse up tree, looking for least common ancestor:
 - Dom[8] \cap Dom[7] = Dom[4]



• See: "A Simple, Fast Dominance Algorithm" Cooper, Harvey, and Kennedy

Completing Control-flow Analysis

- Dominator analysis identifies *back edges*:
 - Edge n \rightarrow h where h dominates n
- Each back edge has a *natural loop*:
 - h is the header
 - All nodes reachable from h that also reach n without going through h
- For each back edge $n \rightarrow h$, find the natural loop:
 - $\{n' \mid n \text{ is reachable from } n' \text{ in } G \{h\}\} \cup \{h\}$
- Two loops may share the same header: merge them
- Nesting structure of loops is determined by set inclusion
 - Can be used to build the control tree





Example Natural Loops



Control Tree:



The control tree depicts the nesting structure of the program.

Natural Loops

Uses of Control-flow Information

- Loop nesting depth plays an important role in optimization heuristics.
 Deeply nested loops pay off the most for optimization.
- Need to know loop headers / back edges for doing
 - loop invariant code motion
 - loop unrolling
- Dominance information also plays a role in converting to SSA form
 - Used internally by LLVM to do register allocation.

Phi nodes Alloc "promotion" Register allocation

REVISITING SSA

Zdancewic CIS 4521/5521: Compilers

Single Static Assignment (SSA)

- LLVM IR names (via %uids) all intermediate values computed by the program.
- It makes the order of evaluation explicit.
- Each %uid is assigned to only once
 - Contrast with the mutable quadruple form
 - Note that dataflow analyses had these kill[n] sets because of updates to variables...
- Naïve implementation of backend: map **%uids** to stack slots
- Better implementation: map %uids to registers (as much as possible)
- Question: How do we convert a source program to make maximal use of %uids, rather than alloca-created storage?
 - two problems: control flow & location in memory
- Then: How do we convert SSA code to x86, mapping %uids to registers?
 - Register allocation.

Alloca vs. %UID

• Current compilation strategy:



- %x = alloca i64 %y = alloca i64 store i64* %x, 3 store i64* %y, 0 %x1 = load %i64* %x %tmp1 = add i64 %x1, 1 store i64* %x, %tmp1 %x2 = load %i64* %x %tmp2 = add i64 %x2, 2 store i64* %y, %tmp2
- Directly map source variables into %uids?



• Does this always work?

What about If-then-else?

• How do we translate this into SSA?





• What do we put for ???

Phi Functions

- Solution: φ functions
 - Fictitious operator, used only for analysis
 - implemented by Mov at x86 level
 - Chooses among different versions of a variable based on the path by which control enters the phi node.

%uid = phi <ty> v_1 , <label₁>, ..., v_n , <label_n>



Phi Nodes and Loops

- Importantly, the **%uids** on the right-hand side of a phi node can be defined "later" in the control-flow graph.
 - Means that **%uids** can hold values "around a loop"
 - Scope of %uids is defined by dominance

```
entry:
  %y1 = ...
  %x1 = ...
  br label %body
body:
  %x2 = phi i64 %x1, %entry, %x3, %body
  %x3 = add i64 %x2, %y1
  %p = icmp slt i64, %x3, 10
  br i1 %p, label %body, label %after
after:
...
```

Alloca Promotion

- Not all source variables can be allocated to registers
 - If the address of the variable is taken (as permitted in C, for example)
 - If the address of the variable "escapes" (by being passed to a function)
- An alloca instruction is called promotable if neither of the two conditions above holds

```
entry:
%x = alloca i64 // %x cannot be promoted
%y = call malloc(i64 8)
%ptr = bitcast i8* %y to i64**
store i65** %ptr, %x // store the pointer into the heap
```

entry:

```
%x = alloca i64  // %x cannot be promoted
%y = call foo(i64* %x) // foo may store the pointer into the heap
```

- Happily, most local variables declared in source programs are promotable
 - That means they can be register allocated

Converting to SSA: Overview

- Start with the ordinary control flow graph that uses allocas
 - Identify "promotable" allocas
- Compute dominator tree information
- Calculate def/use information for each such allocated variable
- Insert ϕ functions for each variable at necessary "join points"
- Replace loads/stores to alloc'ed variables with freshly-generated %uids
- Eliminate the now unneeded load/store/alloca instructions.

Where to Place **\ophi** functions?

- Need to calculate the "Dominance Frontier"
- Node A *strictly dominates* node B if A dominates B and $A \neq B$.
 - Note: A does not strictly dominate B if A does not dominate B or A = B.
- The *dominance frontier* of a node B is the set of all CFG nodes y such that B dominates a predecessor of y but does not strictly dominate y
 - Intuitively: starting at B, there is a path to y, but there is another route to y that does not go through B
- Write DF[n] for the dominance frontier of node n.

Dominance Frontiers

- Example of a dominance frontier calculation results
- $DF[1] = \{1\}, DF[2] = \{1,2\}, DF[3] = \{2\}, DF[4] = \{1\}, DF[5] = \{8,0\}, DF[6] = \{8\}, DF[7] = \{7,0\}, DF[8] = \{0\}, DF[9] = \{7,0\}, DF[0] = \{\}$



Algorithm For Computing DF[n]

- Assume that doms[n] stores the dominator tree (so that doms[n] is the *immediate dominator* of n in the tree)
- Adds each B to the DF sets to which it belongs

```
for all nodes B

if \#(pred[B]) \ge 2 // (just an optimization)

for each p \in pred[B] {

runner := p // start at the predecessor of B

while (runner \neq doms[B]) // walk up the tree adding B

DF[runner] := DF[runner] U {B}

runner := doms[runner]

}
```

Insert ϕ **at Join Points**

- Lift the DF[n] to a set of nodes N in the obvious way: $DF[N] = U_{n \in N} DF[n]$
- Suppose that at variable x is defined at a set of nodes N.

 $\begin{array}{l} \mathsf{DF}_0[\mathsf{N}] = \mathsf{DF}[\mathsf{N}] \\ \mathsf{DF}_{i+1}[\mathsf{N}] = \mathsf{DF}[\mathsf{DF}_i[\mathsf{N}] \cup \mathsf{N}] \end{array}$

```
Let J[N] be the least fixed point of the sequence:

DF_0[N] \subseteq DF_1[N] \subseteq DF_2[N] \subseteq DF_3[N] \subseteq ...

That is, J[N] = DF_k[N] for some k such that DF_k[N] = DF_{k+1}[N]
```

- J[N] is called the "join points" for the set N
- We insert ϕ functions for the variable x at each node in J[N].
 - $x = \phi(x, x, ..., x)$; (one "x" argument for each predecessor of the node)
 - In practice, J[N] is never directly computed, instead you use a worklist algorithm that keeps adding nodes for DF_k[N] until there are no changes, just as in the dataflow solver.
- Intuition:
 - If N is the set of places where x is modified, then DF[N] is the places where phi nodes need to be added, but those also "count" as modifications of x, so we need to insert the phi nodes to capture those modifications too...

Example Join-point Calculation

- Suppose the variable x is modified at nodes 3 and 6
 - Where would we need to add phi nodes?
- $\mathsf{DF}_0[\{3,6\}] = \mathsf{DF}[\{3,6\}] = \mathsf{DF}[3] \cup \mathsf{DF}[6] = \{2,8\}$
- $\mathsf{DF}_1[\{3,6\}]$
 - $= \mathsf{DF}[\mathsf{DF}_0\{3,6\} \cup \{3,6\}]$
 - $= DF[\{2,3,6,8\}]$
 - = DF[2] U DF[3] U DF[6] U DF[8]
 - $= \{1,2\} \cup \{2\} \cup \{8\} \cup \{0\} = \{1,2,8,0\}$
- $\mathsf{DF}_2[\{3,6\}]$
 - $= \dots$ = {1,2,8,0}
- So J[{3,6}] = {1,2,8,0} and we need to add phi nodes at those four spots.

Join Points Pictorially

- Suppose variable x is modified at nodes {3, 6}
 - frontend compiles variable x to "xptr" and uses alloca (in entry node E)



Join Points Pictorially

- Suppose variable x is modified at nodes {3, 6}
 - the frontend compiles variable x to "xptr" and uses alloca (in node 0)
- $\mathsf{DF}_2[\{3,6\}] = \{1,2,8,0\}$



Rename & Insert Phi Nodes

- Suppose variable x is modified at nodes {3, 6}
 - the frontend compiles variable x to "xptr" and uses alloca (in node 0)
- $\mathsf{DF}_2[\{3,6\}] = \{1,2,8,0\}$



Rename & Insert Phi Nodes

• Loads of the original x variable become uses of the renamed version in scope (i.e. the definition that dominates the use)



Phi Placement Alternative

- Less efficient, but easier to understand:
- Place phi nodes "maximally" (i.e. at every node with > 2 predecessors)
- If all values flowing into phi node are the same, then eliminate it:
 %x = phi t %y, %pred1 t %y %pred2 ... t %y %predK
 // code that uses %x

 // code with %x replaced by %y
- Interleave with other optimizations
 - copy propagation
 - constant propagation
 - etc.

Legend of "simple" optimizations*: LAS = load after store LAA = load after alloca DSE = dead store elimination DAE = dead alloca elimination

*nomenclature taken from LLVM IR passes



- How to place phi nodes without breaking SSA?
 - Note: the "real" implementation combines many of these steps into one pass.
 - Places phis directly at the dominance frontier
 - This example also illustrates other common optimizations:
 - Load after store/alloca
 - Dead store/alloca elimination



• How to place phi nodes without breaking SSA?

Insert

 Loads at the end of each block



• How to place phi nodes without breaking SSA?

Insert

- Loads at the end of each block
- Insert φ-nodes at each block



 How to place phi nodes without breaking SSA?

Insert

- Loads at the end of each block
- Insert φ-nodes at each block
- Insert stores after φ-nodes



- For loads after stores (LAS):
 - Substitute all uses of the load by the value being stored
 - Remove the load



- For loads after stores (LAS):
 - Substitute all uses of the load by the value being stored

Remove the load



- For loads after stores (LAS):
 - Substitute all uses of the load by the value being stored

Remove the load



- For loads after stores (LAS):
 - Substitute all uses of the load by the value being stored

Remove the load



- For loads after stores (LAS):
 - Substitute all uses of the load by the value being stored

Remove the load



- For loads after stores (LAS):
 - Substitute all uses of the load by the value being stored
 - Remove the load



- For loads after stores (LAS):
 - Substitute all uses of the load by the value being stored

Remove the load



- Dead Store Elimination (DSE)
 - Eliminate all stores with no subsequent loads.
- Dead Alloca Elimination (DAE)
 - Eliminate all allocas with no subsequent loads/stores.



- Dead Store Elimination (DSE)
 - Eliminate all stores with no subsequent loads.
- Dead Alloca Elimination (DAE)
 - Eliminate all allocas with no subsequent loads/stores.



- Eliminate ϕ nodes:
 - Singletons
 - With identical values from each predecessor
 - See Aycock & Horspool, 2002



- Eliminate **φ** nodes:
 - Singletons
 - With identical values from each predecessor



LLVM Phi Placement

- This transformation is also sometimes called register promotion
 - older versions of LLVM called this "mem2reg" memory to register promotion
- In practice, LLVM combines this transformation with *scalar replacement of aggregates* (SROA)
 - i.e. transforming loads/stores of structured data into loads/stores on register-sized data
- These algorithms are (one reason) why LLVM IR allows annotation of predecessor information in the .ll files
 - Simplifies computing the DF

COMPILER VERIFICATION

Zdancewic CIS 4521/5521: Compilers

LLVM Compiler Infrastructure



Other LLVM IR Features

- C-style data values
 - ints, structs, arrays, pointers, vectors
- Type system
 - used for layout/alignment/padding
- Relaxed-memory concurrency primitives
- Intrinsics
 - extend the language malloc, bitvectors, etc.
- Transformations & Optimizations

Make targeting LLVM IR easy and attractive for developers!



But... it's complex

C O M P						
LLVM Home Documentation»						
113/04/1-000	Abo_initix Type Pointer Type					
LLVM Lang	LLVM Lang • Vector Type					
	 Label Type Token Type 	'llvm.loop.unr	oll_and_jam'			
 Abstra 	 Metadata Ty 	· 'llvm.loop.unr	oll_and_jam.count' Metadata			
Introd	 Aggregate T 	 'llvm.loop.unr 'llvm.loop.unr 	oll_and_jam.disable' Metadat			
• We	Array Typ Structure	 'llvm.loop 	 'fptosi to' Instru 			
 Identif 	Opaque	'llvm.loop	 Syntax: 			
 High L 	Constants	 'llvm.mem' 	 Arguments: 			
• Mo	 Simple Constar Complex Const 	· 'irr loop'	 Semantics: 			
• Lin	Global Variable	 'invariant 	Example:			
• Cal	 Undefined Valu 	 'type' Meta 	Syntax:			
• Vis	Poison Values Addresses of P	 'associate 	 Overview: 			
• DLI	 Constant Expre 	branch	 Arguments: 			
• Thr	Other Values	 function 	 Semantics: Example: 			
• Rur	 Inline Assemble 	VP Module Floor Mo	 'sitofp to' Instru 			
• Stri	Output c	 Objective-C G 	Syntax:			
• Noi	Input cor	Metadata	 Overview: Arguments: 			
• Glo	 Indirect i 	 C type width I Automatic Linker 	 Semantics: 			
• Fur	Constrair	ThinLTO Summar	 Example: 			
• Alia	 Supporte 	Module Path S	Syntax:			
• IFu	 Asm templa 	Global Value : Function Si	 Overview: 			
• Cor	Metadata	 Global Vari 	 Arguments: 			
• Nar	Metadata Node	Alias Sumn Eusetion El	 Semantics: Example: 			
• Par	 Specialized 	Calls	 'inttoptr to' Inst 			
• Gar	 DICompil DIFile 	 Refs 	 Syntax: 			
• Pre	 DIBasicTy 	TypeTes	 Overview: Arguments: 			
• Pro	 DISubrou 	 TypeTes 	 Semantics: 			
• Per	 DiDerived DiCompo 	 TypeCh TypeTcr 	Example:			
• Att	 DISubran 	 TypeTe: TypeCh 	Syntax:			
• Fur	 DIEnume DITemple 	Type ID Summ	 Overview: 			
• Glo	 DiTempla 	 Intrinsic Global V The 'llvm.use 	 Arguments: Somentics: 			
• Op	 DINames 	• The 'llvm.com	 Example: 			
•	 DIGlobal DISubara 	• The 'llvm.glo	 'addrspacecast t 			
•	 Disubpro DiLexical 	The 'llvm.glc Instruction Refer	 Syntax: 			
	 DILexical 	Terminator In	 Arguments: 			
• Mo	 DILocatic DILocalV 	 'ret' Instru 	Semantics:			
• Dat	 DIExpres 	 Syntax: Overvier 	 Example: Other Operations 			
• Tar	 DIObjCPr 	 Argume 	'icmp' Instruction			
POI	 Dilmport DiMacro 	 Semanti Example 	Syntax:			
	 DIMacroF 	 'br' Instruc 	 Overview: Arguments: 			
	 'tbaa' Metac 	• Syntax:	 Semantics: 			
O Ato	Semantic Represent	Overvie Argume	Example:			
o Fid	 'tbaa.struc 	 Semanti 	Syntax:			
• Fas	 'noalias' an 	Example	 Overview: 			
O USE	 'fpmath' Met 	Svntax	 Arguments: 			
Sol	"range" Meta	Overvier	 Semantics: Example: 			
- Type s	 'callees' Mo 	Argume Semanti	 'phi' Instruction 			
	'unpredicta	 Implem 	Syntax:			
o Fur	'llvm.loop'	Example	 Overview: Arguments: 			
FILS	 'llvm.loop. 	Indirectb Syntax	 Semantics: 			
-	 'llvm.loop. 	Overvier	Example:			
	'llvm.loop.	Argume	 Select Instruction Syntax: 			
	'llvm.loop.	 Implem 	 Overview: 			
	· 'llvm.loop.	 Example 	 Arguments: Semantics: 			
	 'llvm.loop. 	'invoke' in Syntax	 Example: 			
	 '11vm.loop. 	Overvier	 'call' Instruction 			

 Arguments: 		
 Semantics: 		
 Example: 		
 'catchswitch' Instruction 	 'unem' Instruction 	ì
Syntax:	Syntax:	
 Overview; 	 Overview: 	
 Arguments: 	 Arguments: 	
 Semantics: 	 Semantics: 	
 Example: 	 Example: 	
 'catchret' Instruction 	 'srem' Instruction 	ı
 Svntax: 	 Syntax: 	
 Overview: 	 Overview: 	
 Arguments: 	 Arguments: 	
 Semantics: 	 Semantics: 	
 Example: 	 Example: 	
 'cleanupret' Instruction 	 'frem' Instruction 	ì
Syntax:	Syntax:	
 Overview; 	 Overview: 	
 Arguments: 	 Arguments: 	
 Semantics: 	 Semantics: 	
 Example: 	 Example: 	
 'unreachable' Instruction 	 Bitwise Binary Operation 	ı
 Svntax: 	 'sh1' Instruction 	
 Overview: 	Syntax:	
 Semantics: 	 Overview: 	
Binary Operations	 Arguments: 	
 'add' Instruction 	 Semantics: 	
 Svntax: 	 Example: 	
 Overview: 	 '1shr' Instruction 	1
 Arguments: 	Syntax:	
	Dofe	
LLVIV	I KEIE	
+ -		

'fdiv' Instruction

Overview:

Arguments:

Semantics:

Example:

human! Instruct

Svntax:

 Example: Arguments: Syntax: Overview: Syntax: Overview: Syntax: Syntax:<!--</th--><th>w: w: ics: b: b: b: to' w: w: ics: b: cs: b: ics: e: ics: e: ics: e: ics: ics: ics: b: ics: b: b: b: b: b: b: b: b: b: b</th>	w: w: ics: b: b: b: to' w: w: ics: b: cs: b: ics: e: ics: e: ics: e: ics: ics: ics: b: ics: b: b: b: b: b: b: b: b: b: b

Arguments:

Semantics:

'extractvalue' Inst

Example:

Aggregate Operation

Syntax:

Overview:

Arguments:
 Semantics:

Example:

_		
	 'fptosi to' in 	struction
	Syntax:	Intrinsic Functi
	 Overview: 	Variable Arc
commpres	 Argument 	• '11vm.va
Memory Access a	Semantics	Sunta
 'alloca' Instri 	Example:	- Oyana
 Syntax: 	'uitofp t	- Overv
 Overview: 	Syntax:	- Argui
 Arguments 	 Overview: 	Jenia (11) marcina
 Semantics: 	Argument	• 11vm.va
 Example: 	 Semantics 	 Synta
 'load' Instruct 	 Example: 	 Overv
 Syntax: 	 feitofn t 	 Argur
 Overview: 	Sitterp	 Sema
 Arguments 	• Syntax:	'llvm.va
 Semantics: 	 Overview: 	 Synta
Examples:	 Argument 	 Overv
 'store' Instru 	 Semantics 	 Argur
Syntax:	 Example: 	 Sema
 Overview: 	 'ptrtoint 	 Accurate Ga
 Arguments 	Syntax:	 Experime
 Semantics: 	 Overview: 	'llvm.gc
 Example: 	 Argument 	Synta
 'fence' Instru 	 Semantics 	Overv
Syntax:	Example:	Arour
 Overview: 	inttoptr	 Sema
 Arguments 	Syntax:	 '11vm.gc
 Semantics: 	Overview	II viii. ge

<u>eference Manual</u> onto

Memory Acc

'alloca'

(0)	пеш	
<u> </u>		
	, a gamen	Argur
Syntax:	 Semantics 	 Sema
 Overview: 	 Example: 	'llvm.ad
 Arguments 	 Other Operation 	 Synta
 Semantics: 	 'icmp' Instru 	 Overv
 Example: 	Syntax:	 Sema
 Vector of p 	 Overview: 	• 'llvm.fr
Conversion Oper	 Argument 	 Svnta
• trunc to	 Semantics 	Overv
 Syntax: 	 Example: 	Argur
Overview:	 'fcmp' Instru 	 Sema
- Arguments	Syntax:	 'llvm.lo
 Example: 	 Overview: 	Intrinsics
'zext to'l	Argument	Synta
Syntax:	 Semantics 	Overv
Overview:	 Example: 	Arour
 Arguments 	'phi' Instruct	 Sema
 Semantics: 	 Syntax: 	 '11vm.re
Example:	 Overview: 	Intrinsics
• 'sext to'	Argument	Sunta
Syntax:	 Semantics 	Overu
 Overview: 	Example:	Sema
 Arguments 	 'select' Inst 	'llym.st
Semantics:	Syntax:	Cunto
Example:	Overview	- Synta Ovoru
 'fptrunc 1 	Argument	Somo
Syntax:	Semantics	'llum et
 Overview: 	Example:	Sunta
 Arguments 	 'call' instruit 	Over
 Semantics: 	Syntax:	- Overv
 Example: 	Overview	Jahren de
 'tpext to' 	Argument	Livm.ge
 Syntax: 	Semantics	Syrita
Overview:	Example:	- Overv
- Arguments	 'va ang' inst 	Sema 111vc - co
 Semantics: Example: 	• Suntax:	- '11vm.pr
'fntoui tr	- Syntax.	• Synta
- Suntav:	- Argumont	Overv
 Overview: 	- Argumen	Argur
 Arguments: 		 Sema
, a aumento.		

		/		Z			Ĵ	
-	11.1	and a second			1		£.	
-	Intrinsic	(perimental.vector.red)	 'llvm.expe 	riment	tal.veo	tor.reduce.and.*'		
	 Syn 	· · · · · · · · · · · · · · · · · · ·	intrinsic					
	• Ove	Suntax:	uncer manisic					
	Arg	• Overview:						
•	'llvm.	 Semantics: 				r.reduce.or.*'		
	Intrins	'llvm.clear_cache	' Intrinsic					
	Syn	 Syntax: 						
	Arc	Overview:						
	'llvm.	· 11vm.instrorof.i	ncrement' Intrinsic			r.reduce.xor.*'		
	Intrins	 Syntax: 						
	 Syn 	 Overview: 				'llvm.load.relative	Intrin	nsic
	• Ove	 Arguments: 				Syntax:		
	• Arg	Semantics:				 Overview: 		
-	Intrins	Suptax:	ncrement.step inti		•	'llvm.sideeffect' Inf	rinsic	
	Syn	 Overview: 				Syntax:		
	• Ove	 Arguments: 				 Overview: 		
	Arg	 Semantics: 				Arguments:		
•	'llvm.	 'llvm.instrprof.v 	alue.profile' Intri			Semantics:		
	Intrins	 Syntax: 			• St	ack Map Intrinsics		
	- Syn	Overview: Arguments:			o El	ement Wise Atomic Me	mory	Intrinsics
	Arc	 Semantics: 			•	'llvm.memcpy.elemen	t.unor	rdered.atomic'
<u> </u>	'llvm.	'llvm.thread.poir	ter' Intrinsic			Intrinsic		
	Intrins	Syntax:				Syntax:		
	 Syn 	 Overview: 				 Overview: 		
	• Ove	 Semantics: Standard C Library Int 	elacies			Arguments:		
	Arg	 Standard C Library Ini '11vm.memcny' Intri 	nsic			Semantics:		
10	Intrine	 Syntax: 				Lowering:		
	• Svn	 Overview: 			•	'llvm.memmove.eleme	nt.und	ordered.atomic'
	• Ove	 Arguments: 				Intrinsic		
	Arg	 Semantics: 				Syntax:		
11	'llvm.	'IIVm.memmove' Int	rinsic			 Overview: 		
	Intrins	 Overview: 				 Arguments: 		
	- Syn	 Arguments: 				 Semantics: 		
	Arc	 Semantics: 				Lowering:		
	'llvm.	 'llvm.memset.*' in 	trinsics			'llvm.memset.elemen	t.unor	rdered.atomic'
	Intrins	Syntax:				Intrinsic		
	 Syn 	 Arguments: 				Syntax:		
	 Ove 	 Semantics: 				 Overview: 		
0 40	Arg	 'llvm.sqrt.*' intri 	nsic			 Arguments: 		
110	'11vm.	 Syntax: 				Semantics:		
	Syn	Overview:				Lowering:		
	• Ove	 Arguments: Semantics: 						
	Arg	 'llvm.powi.*' Intri 	nsic	Abs	stract			
	 Sen 	Syntax:		-				c
-	 Exa 	 Overview: 		Th	is doc	ument is a reference n	anual	for the LLVM assembly
	Svn	 Arguments: 		lar	nguag	e. LLVM is a Static Sing	le Ass	ignment (SSA) based
	Ove	Semantics:	ele	rep	oreser	itation that provides ty	pe sat	ety, low-level opera-
	 Arg 	Syntax:	SIC	tio	ns, fie	exibility, and the capab	ility of	representing all
	 Sen 	 Overview: 						
_	 Exa 	 Arguments: 						
• De	bugger	 Semantics: 						
o Tr	ampoli	 'llvm.cos.*' Intrin 	SIC					
	'11vm.	 Syntax: Overview: 				trinsic		
	 Syn 	 Arguments: 						
	• Ove	 Semantics: 						
	• Arg	'llvm.pow.*' Intrin	sic					
	 Sen '11vm 	Syntax:				Intellector		
	Svn	Overview: Arguments:				intrinsic		
	 Ove 	 Semantics: 						

Overview:

Semantics:

Syntax:

Overview

Arguments:

Syntax:

'llvm.exp.*' Intrinsic

'llvm.exp2.*' Intrinsic

Arg
 Sen

Syn
Ove
Arg

Masked V

'llvm.

Intrinsics nsics

One Example: undef

The **undef** "value" represents an arbitrary, but indeterminate bit pattern for any type.

Used for:

- uninitialized registers
- reads from volatile memory
- results of some underspecified operations

What is the value of **%y** after running the following?

One plausible answer: 0 Not LLVM's semantics!

(LLVM is more liberal to permit more aggressive optimizations)

Partially defined values are interpreted *nondeterministically* as sets of possible values:

Interactions with Optimizations

Consider:

versus:



Interactions with Optimizations

Consider:

versus:

%y = add i8 %x, %x

Upshot: if %x is **undef**, we can't optimize **mul** to **add** (or vice versa)!

What's the problem?

Bug List: (12 of 435) First Last Prev Next Show last search results				
Bug 33165 - Simpify* cannot distribute instructions for simplification due to under	f			
	Benerted: 2017-05-25 02:12 DDT by Nune Longs			
Davide Italiano 2017-05-25 08:55:40 PDT	Comment 6			
We				
cc Davide Italiano 2017-05-25 09:05:26 PDT	Comme			
Тс				
n (unless we want to give up on some undef trans	formations, and special case sel(
B' John Regehr 2017-05-25 09:09:24 PDT	<u>Com</u> ı			
d Yes, this is one of those test cases. There are so many optimization failures Nuno has been automatically filtering out classes of mistranslation that are to be hard to fix but I guess he decided to take a closer look at some of the				
Soon I'll be able to include branches/phis is branches due to a limitation in Alive.	n these test cases, but only forw			

Compiler Bugs

[Regehr's group: Yang et al. PLDI 2011]



LLVM is hard to trust (especially for critical code)

What can we do about it?

Approaches to Software Reliability

- Social
 - Code reviews
 - Extreme/Pair programming
- Methodological
 - Design patterns
 - Test-driven development
 - Version control
 - Bug tracking
- Technological
 - "lint" tools, static analysis
 - Fuzzers, random testing
- Mathematical
 - Sound programming languages tools
 - "Formal" verification

Less "formal": Techniques may miss problems in programs

This isn't a tradeoff... all of these methods should be used.

Even "formal" methods can have holes:

- Did you prove the right thing?
- Do your assumptions match reality?
- Knuth. "Beware of bugs in the above code; I have only proved it correct, not tried it."



Goal: Verified Software Correctness

- Social
 - Code reviews
 - Extreme/Pair programming
- Methodological
 - Design patterns
 - Test-driven development
 - Version control
 - Bug tracking
- Technological
 - "lint" tools, static analysis
 - Fuzzers, random testing
- Mathematical
 - Sound programming languages tools
 - "Formal" verification

Q: How can we move the needle towards mathematical software correctness properties?

Taking advantage of advances in computer science:

- Moore's law
- improved programming languages
 & theoretical understanding
- better tools: interactive theorem provers

CompCert – A Verified C Compiler



Xavier Leroy INRIA Optimizing C Compiler, proved correct end-to-end with machine-checked proof in Coq



Csmith on CompCert?



Verification Works!

"The striking thing about our CompCert results is that the middle-end bugs we found in all other compilers are absent. As of early 2011, the under-development version of CompCert is the only compiler we have tested *for which Csmith cannot find wrong-code errors*. This is not for lack of trying: we have <u>devoted about six CPU-years</u> to the task. *The apparent unbreakability of CompCert supports a strong argument that developing compiler optimizations within a proof framework, where safety checks are explicit and machine-checked, has tangible benefits for compiler users."*

- Regehr et. al 2011

Our Approach: Formal Verification

Interactive theorem proving in Coq

- not model checking / SMT
- human-in-the-loop



Using Coq *is* functional programming ...but some of your programs *are* proofs

 \Rightarrow proof engineering

The Vellvm Project

[Zhao et al. POPL 2012, CPP 2012, PLDI 2013, Zackowski, et al. ICFP2021]



- Formal semantics
- Facilities for creating simulation proofs
- Implemented in Coq
- Extract passes for use with LLVM compiler
- Example: verified memory safety instrumentation