Announcements

• HW6: Analysis & Optimizations
  – Alias analysis, constant propagation, dead code elimination, register allocation
  – Due: TOMORROW at Midnight

• Final Exam:
  – Date: Friday, May 4th
  – Time: noon – 2:00pm
  – Location: Moore 216
  – Coverage: cumulative, but emphasis on material since the midterm
Phi nodes
Alloc “promotion”
Register allocation

REVISITING SSA
• 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
Phi Functions

• Solution: $\phi$ 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.
    \[
    \%uid = \phi <ty> \ v_1, <label_1>, \ldots, v_n, <label_n>
    \]

```c
int y = ...
int x = ...
int z = ...
if (p) {
  x = y + 1;
} else {
  x = y * 2;
}
z = x + 3;
```

entry:
```c
  %y1 = ...
  %x1 = ...
  %z1 = ...
  %p = icmp ...
  br il %p, label %then, label %else
then:
  %x2 = add i64 %y1, 1
  br label %merge
else:
  %x3 = mult i64 %y1, 2
merge:
  %x4 = phi i64 %x2, %then, %x3, %else
  %z2 = %add i64 %x4, 3
```
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
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 = \text{phi } t \%y, \%\text{pred1 } t \%y \%\text{pred2 } ... t \%y \%\text{predK}\n  // code that uses \%x
  ⇒
  // code with \%x replaced by \%y

• Interleave with other optimizations
  – copy propagation
  – constant propagation
  – etc.
Example SSA Optimizations

- 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/allocation
  - Dead store/allocation elimination
Example SSA Optimizations

• How to place phi nodes without breaking SSA?
  - Insert Loads at the end of each block

```
\text{l_1: } \%p = \text{alloca i64} \\
\text{store 0, } \%p \\
\%b = \%y > 0 \\
\text{\%x_1 = load } \%p \\
\text{br } \%b, \%l_2, \%l_3

\text{l_2: } \\
\text{store 1, } \%p \\
\text{\%x_2 = load } \%p \\
\text{br } \%l_3

\text{l_3: } \\
\%x = \text{load } \%p \\
\text{ret } \%x
```
Example SSA Optimizations

- How to place phi nodes without breaking SSA?
- Insert
  - Loads at the end of each block
  - Insert φ-nodes at each block
Example SSA Optimizations

• How to place phi nodes without breaking SSA?
  - Insert
    - Loads at the end of each block
    - Insert $\phi$-nodes at each block
    - Insert stores after $\phi$-nodes

l₁:  %p = alloca i64
    store 0, %p
    %b = %y > 0
    %x₁ = load %p
    br %b, %l₂, %l₃

l₂:  %x₃ = $\phi$[%x₁,%l₁]
    store %x₃, %p
    store 1, %p
    %x₂ = load %p
    br %l₃

l₃:  %x₄ = $\phi$[%x₁,%l₁, %x₂,%l₂]
    store %x₄, %p
    %x = load %p
    ret %x

Find alloca
max $\phi$s
LAS/LAA
DSE
DAE
elim $\phi$s
Example SSA Optimizations

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

```c
l_1: %p = alloca i64
    store 0, %p
    %b = %y > 0
    %x_1 = load %p
    br %b, %l_2, %l_3

l_2: %x_3 = phi[%x_1, %l_1]
    store %x_3, %p
    store 1, %p
    %x_2 = load %p
    br %l_3

l_3: %x_4 = phi[%x_1, %l_1, %x_2, %l_2]
    store %x_4, %p
    %x = load %p
    ret %x
```
Example SSA Optimizations

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

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

```
l_1: %p = alloca i64
    store 0, %p
    %b = %b > 0
    %x_1 = load %p
    br %b, %l_2, %l_3

l_2: %x_3 = phi[0,%l_1]
    store %x_3, %p
    store 1, %p
    %x_2 = load %p
    br %l_3

l_3: %x_4 = phi[0;%l_1, %x_2;%l_2]
    store %x_4, %p
    %x = load %p
    ret %x
```
Example SSA Optimizations

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

```c
l_1: %p = alloca i64
    store 0, %p
    %b = %y > 0
    br %b, %l_2, %l_3

l_2: %x_3 = phi[0, %l_1]
    store %x_3, %p
    store 1, %p
    %x_2 = load %p
    br %l_3

l_3: %x_4 = phi[0; %l_1, %x_2, %l_2]
    store %x_4, %p
    %x = load %p
    ret %x
```
Example SSA Optimizations

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

```
l_1: %p = alloca i64
    store 0, %p
    %b = %y > 0
    br %b, %l_2, %l_3

l_2: %x_3 = φ[0,%l_1]
    store %x_3, %p
    store 1, %p
    %x_2 = load %p
    br %l_3

l_3: %x_4 = φ[0;%l_1, 1;%l_2]
    store %x_4, %p
    %x = load %p
    ret %x
```
Example SSA Optimizations

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

```
l_1: %p = alloca i64
    store 0, %p
    %b = %y > 0
    br %b, %l_2, %l_3

l_2: %x_3 = phi[0, %l_1]
    store %x_3, %p
    store 1, %p
    br %l_3

l_3: %x_4 = phi[0; %l_1, 1; %l_2]
    store %x_4, %p
    %x = load %p
    ret %x
```
Example SSA Optimizations

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

```
%p = alloca i64
store 0, %p
%b = %y > 0
br %b, %l2, %l3

%x3 = phi [0, %l1]
store %x3, %p
store 1, %p
br %l3

%x4 = phi [0: %l1, 1: %l2]
store %x4, %p
%x = load %p
ret %x4
```
Example SSA Optimizations

- **Dead Store Elimination (DSE)**
  - Eliminate all stores with no subsequent loads.

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

```
\[ l_1: \%p = \text{alloca i64} \]
\[ \text{store 0, } \%p \]
\[ \%b = \%y > 0 \]
\[ \text{br } \%b, \%l_2, \%l_3 \]

\[ l_2: \%x_3 = \phi[0, \%l_1] \]
\[ \text{store } \%x_3, \%p \]
\[ \text{store 1, } \%p \]
\[ \text{br } \%l_3 \]

\[ l_3: \%x_4 = \phi[0; \%l_1, 1; \%l_2] \]
\[ \text{store } \%x_4, \%p \]
\[ \text{ret } \%x_4 \]
Example SSA Optimizations

- **Dead Store Elimination (DSE)**
  - Eliminate all stores with no subsequent loads.

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

Source code:

```
  l1: %p = alloca i64
      store 0, %p
      %b = %y > 0
      br %b, %l2, %l3

  l2: %x3 = φ[0, %l1]
      store %x3, %p
      store 1, %p
      br %l3

  l3: %x4 = φ[0; %l1, 1:%l2]
      store %x4, %p
      ret %x4
```

Diagram:

1. Find alloca
2. max φs
3. LAS/LAA
4. DSE
5. DAE
6. elim φs
7. Dead Store Elimination (DSE)
8. Dead Alloca Elimination (DAE)
Example SSA Optimizations

\[ l_1: \%
\text{b} = \%
\text{y} > 0 \]
\[ \text{br} \%
\text{b}, \%
\text{l}_2, \%
\text{l}_3 \]

\[ l_2: \%
\text{x}_3 = \phi[0,\%
\text{l}_1] \]
\[ \text{br} \%
\text{l}_3 \]

\[ l_3: \%
\text{x}_4 = \phi[0;\%
\text{l}_1, 1;\%
\text{l}_2] \]
\[ \text{ret} \%
\text{x}_4 \]

- Eliminate \( \phi \) nodes:
  - Singletons
  - With identical values from each predecessor
  - See Aycock & Horspool, 2002
Example SSA Optimizations

Example SSA Optimizations

\[ l_1: \]
\[ \%b = \%y > 0 \]
\[ \text{br } \%b, \%l_2, \%l_3 \]

\[ l_2: \%x_3 = \phi[0,\%l_1] \]
\[ \text{br } \%l_3 \]

\[ l_3: \%x_4 = \phi[0;\%l_1, 1:\%l_2] \]
\[ \text{ret } \%x_4 \]

- Eliminate \( \phi \) nodes:
  - Singletons
  - With identical values from each predecessor
Example SSA Optimizations

l₁:

%b = %y > 0
br %b, %l₂, %l₃

l₂:

br %l₃

l₃: %x₄ = φ[0;%l₁, 1;%l₂]

ret %x₄

Find alloca
max φs
LAS/LAA
DSE
DAE
elim φ

• Done!
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
Random test-case generation

Source Programs

LLVM

64

GCC

79 bugs: 25 critical

202 bugs

325 bugs in total

{8 other C compilers}

Verified Compilation: CompCert [Leroy et al.]
(Not directly applicable to LLVM)
CompCert – A Verified C Compiler

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

Xavier Leroy
INRIA

Diagram:

- C source
- Clight
- C#minor
- Cminor
- RTL
- LTL
- Linear
- Mach
- PPC
- PowerPC assembly
- Coq
- mini-ML
- Other languages?
- Program prover
- Model checker
- State analyzer
- Compiler: CompCert
- ISA

Flow:
- C source → Clight → C#minor → Cminor → RTL → LTL → Linear → Mach → PPC → PowerPC assembly
- Coq
- mini-ML
- Other languages?
- Programmed in Caml
- Programmed and proved in Coq

Notes:
- Optimizing C Compiler, proved correct end-to-end with machine-checked proof in Coq
- Xavier Leroy, INRIA
- Diagram showing the compilation process from C source to PowerPC assembly, with verification steps along the way.
Comparing Behaviors

• Consider two programs $P_1$ and $P_2$ possibly in different languages.
  – e.g. $P_1$ is an Oat program, $P_2$ is its compilation to LL

• The semantics of the languages associate to each program a set of observable behaviors:

  $\mathcal{B}(P)$ and $\mathcal{B}(P')$

• Note: $|\mathcal{B}(P)| = 1$ if $P$ is deterministic, $> 1$ otherwise
What is Observable?

• For C-like languages:

observable behavior ::= 
  | terminates(st) (i.e. observe the final state) 
  | diverges 
  | goeswrong

• For pure functional languages:

observable behavior ::= 
  | terminates(v) (i.e. observe the final value) 
  | diverges 
  | goeswrong
What about I/O?

• Add a trace of input-output events performed:

\[
\begin{align*}
\text{t} & ::= \; [\] \mid e :: t \quad \text{(finite traces)} \\
\text{coind. T} & ::= \; [\] \mid e :: T \quad \text{(finite and infinite traces)}
\end{align*}
\]

observable behavior ::= \\
| terminates(t, st) \quad \text{(end in state st after trace t)} \\
| diverges(T) \quad \text{(loop, producing trace T)} \\
| goeswrong(t)

Examples

• P1:
  `print(1); / st`  \(\Rightarrow\)  `terminates(out(1)::[]),st)`

• P2:
  `print(1); print(2); / st`
  \(\Rightarrow\)  `terminates(out(1)::out(2)::[]),st)`

• P3:
  `WHILE true DO print(1) END / st`
  \(\Rightarrow\)  `diverges(out(1)::out(1)::...)`

• So  \(B(P1) \neq B(P2) \neq B(P3)\)
• Two programs P1 and P2 are bisimilar whenever:

\[ B(P1) = B(P2) \]

• The two programs are completely indistinguishable.

• But… this is often too strong in practice.
Compilation Reduces Nondeterminism

- Some languages (like C) have underspecified behaviors:
  - Example: order of evaluation of expressions \( f() + g() \)

- Concurrent programs often permit nondeterminism
  - Classic optimizations can reduce this nondeterminism
  - Example:
    \[
    \begin{aligned}
    a &:= x + 1; b := x + 1 \quad || \quad x := x+1 \\
    \end{aligned}
    \]
    vs.

    \[
    \begin{aligned}
    a &:= x + 1; b := a \quad || \quad x := x+1 \\
    \end{aligned}
    \]

- LLVM explicitly allows nondeterminism:
  - undef values (not part of LLVM lite)
  - see the discussion later
Backward Simulation

• Program P2 can exhibit fewer behaviors than P1:

\[ \mathbb{B}(P1) \supseteq \mathbb{B}(P2) \]

• All of the behaviors of P2 are permitted by P1, though some of them may have been eliminated.

• Also called refinement.
What about goes wrong?

- Compilers often translate away bad behaviors.

\[
x := 1/y ; x := 42 \quad \text{vs.} \quad x := 42
\]
(divide by 0 error) \hspace{1cm} (always terminates)

- Justifications:
  - Compiled program does not "go wrong" because the program type checks or is otherwise formally verified
  - Or just "garbage in/garbage out"
Safe Backwards Simulation

• Only require the compiled program’s behaviors to agree if the source program could not go wrong:

\[
goeswrong(t) \notin \mathcal{B}(P1) \implies \mathcal{B}(P1) \supseteq \mathcal{B}(P2)
\]

• Idea: let \( S \) be the functional specification of the program:
  A set of behaviors not containing \( \text{goeswrong}(t) \).
  – A program \( P \) satisfies the spec if \( \mathcal{B}(P) \subseteq S \)

• Lemma: If \( P2 \) is a safe backwards simulation of \( P1 \) and \( P1 \) satisfies the spec, then \( P2 \) does too.
**Idea:** The event trace along a (target) sequence of steps originating from a compiled program must correspond to some source sequence.

**Tricky parts:**
- Must consider all possible target steps
- If the compiler uses many target steps for once source step, we have to invent some way of relating the intermediate states to the source.
- The compilation function goes the wrong way to help!
Safe Forwards Simulation

• Source program’s behaviors are a subset of the target’s:

\[ \text{goeswrong}(t) \notin \mathcal{B}(P1) \Rightarrow \mathcal{B}(P1) \subseteq \mathcal{B}(P2) \]

• P2 captures all the good behaviors of P1, but could exhibit more (possibly bad) behaviors.

• But: Forward simulation is significantly easier to prove:
  – Only need to show the existence of a compatible target trace.
Determinism!

• Lemma: If $P_2$ is deterministic then forward simulation implies backward simulation.

• Proof: $\emptyset \subset B(P_1) \subseteq B(P_2) = \{b\}$ so $B(P_1) = \{b\}$.

• Corollary: safe forward simulation implies safe backward simulation if $P_2$ is deterministic.
Forward Simulations

Source: \( \sigma_1 \rightarrow \sigma_2 \)

Target: \( \mathcal{C}[\sigma_1] \rightarrow \tau_2 \rightarrow \tau_3 \rightarrow \cdots \mathcal{C}[\sigma_2] \)

Idea: Show that every transition in the source program:
- is simulated by some sequence of transitions in the target
- while preserving a relation \( \sim \) between the states
The Vellvm Project

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

[Zhao et al. POPL 2012, CPP 2012, PLDI 2013]
Vellvm Framework

Type System and SSA
Operational Semantics
Syntax
Memory Model
Proof Techniques & Metatheory

Extract
OCaml Bindings
Parser
Printer

LLVM
Other Optimizations
Target

C Source Code
LLVM IR
Transform
LLVM IR

Coq
Vellvm Framework

- Type System and SSA
- Operational Semantics
- Syntax
- Memory Model

Proof Techniques & Metatheory

Coq

Extract

Verified Transform

OCaml Bindings

Parser

Printer

C Source Code

LLVM IR

LLVM IR

LLVM

Other Optimizations

Target
Final Exam

- Will mostly cover material since the midterm
  - Starting from Lecture 13
  - Scope / Typechecking / Inference Rules
  - Lambda calculus / closure conversion
  - Objects, inheritance, types, implementation of dynamic dispatch
  - Basic optimizations
  - Dataflow analysis (forward vs. backward, fixpoint computations, etc.)
    - Liveness
  - Graph-coloring Register Allocation
  - Control flow analysis
    - Loops, dominator trees

- Will focus more on the theory side of things
- Format will be similar to the midterm
  - Simple answer, computation, multiple choice, etc.
  - Sample exam from last time is on the web
What have we learned?
Where else is it applicable?
What next?
Why CIS 341?

• You will learn:
  – Practical applications of theory
  – Parsing
  – How high-level languages are implemented in machine language
  – (A subset of) Intel x86 architecture
  – A deeper understanding of code
  – A little about programming language semantics
  – Functional programming in OCaml
  – How to manipulate complex data structures
  – How to be a better programmer

• Did we meet these goals?
Stuff we didn’t Cover

• We skipped stuff at every level…
• Concrete syntax/parsing:
  – Much more to the theory of parsing… LR(*)
  – Good syntax is art not science!
• Source language features:
  – Exceptions, advanced type systems, type inference, concurrency
• Intermediate languages:
  – Intermediate language design, bytecode, bytecode interpreters, just-in-time compilation (JIT)
• Compilation:
  – Continuation-passing transformation, efficient representations, scalability
• Optimization:
  – Scientific computing, cache optimization, instruction selection/optimization
• Runtime support:
  – memory management, garbage collection
• CIS 500: Software Foundations
  – Prof. Pierce
  – Theoretical course about functional programming, proving program properties, type systems, lambda calculus. Uses the theorem prover Coq.

• CIS 501: Computer Architecture
  – Prof. Devietti
  – 371++: pipelining, caches, VM, superscalar, multicore,…

• CIS 552: Advanced Programming
  – Prof. Weirich
  – Advanced functional programming in Haskell, including generic programming, metaprogramming, embedded languages, cool tricks with fancy type systems

• CIS 670: Special topics in programming languages
Where to go from here?

• Conferences (proceedings available on the web):
  – Programming Language Design and Implementation (PLDI)
  – Principles of Programming Languages (POPL)
  – Object Oriented Programming Systems, Languages & Applications (OOPSLA)
  – International Conference on Functional Programming (ICFP)
  – European Symposium on Programming (ESOP)
  – …

• Technologies / Open Source Projects
  – Yacc, lex, bison, flex, …
  – LLVM – low level virtual machine
  – Java virtual machine (JVM), Microsoft’s Common Language Runtime (CLR)
  – Languages: OCaml, F#, Haskell, Scala, Go, Rust, …?
Where else is this stuff applicable?

- **General programming**
  - In C/C++, better understanding of how the compiler works can help you generate better code.
  - Ability to read assembly output from compiler
  - Experience with functional programming can give you different ways to think about how to solve a problem

- **Writing domain specific languages**
  - lex/yacc very useful for little utilities
  - understanding abstract syntax and interpretation

- **Understanding hardware/software interface**
  - Different devices have different instruction sets, programming models
Thanks!

• To the TAs: Richard, Nicolas, Olek, Yishuai
  – for doing an amazing job putting together the projects for the course.

• To you for taking the class!

• How can I improve the course?
  – Let me know in course evaluations!