Common Optimization

1. Constant folding
e.g. x = 3+6 => x =9
2. Strength reduction
e.g. z = y*4 => z = y<<2
3. Common subexpression elimination
e.g.
x = y
z = z + x => z = z+y

Dataflow analysis (DFA)

1. An example of DFA – reaching definition
2. Safe answer: can have more bindings than the "true" answer, but can't miss any (May Analysis)

Lattice Definition

A lattice is a tuple $(S, \sqsubseteq, \bot, \top, \sqcup, \sqcap)$ such that:

- $(S, \sqsubseteq)$ is a poset
- $\forall a \in S . \bot \sqsubseteq a$
- $\forall a \in S . a \sqsubseteq \top$
- Every two elements from S have a lub and a glb
- $\sqcup$ is the least upper bound operator, called a join
- $\sqcap$ is the greatest lower bound operator, called a meet

Reaching Definitons(May Analysis)

```
   ↓ in
S: X := ...
   ↓ out
```

out = in – { X → S' | S' ∈ stmts } ∪ { X → S }

```
   ↓ in
S: *P := ...
   ↓ out
```

- Using may-point-to information:
  out = in ∪ { X → S | X ∈ may-point-to(P) }
- Using must-point-to aswell:
  out = in – { X → S' | X ∈ must-point-to(P) ∧ S' ∈ stmts } ∪ { X → S | X ∈ may-point-to(P) }

Constant Prop (Must Analysis)

```
  ↓ in
X := N
  ↓ out
```

$F_{X := N}(in) = in - \{ X \to * \} \cup \{ X \to N \}$

```
  ↓ in
X := Y op Z
  ↓ out
```

$F_{X := Y \text{ op } Z}(in) = in - \{ X \to * \} \cup \{ X \to N \mid (Y \to N_1) \in in \wedge (Z \to N_2) \in in \wedge N = N_1 \text{ op } N_2 \}$

```
  ↓ in
X := *Y
  ↓ out
```

$F_{X := *Y}(in) = in - \{ X \to * \} \cup \{ X \to N \mid \forall Z \in \text{may-point-to}(Y) . (Z \to N) \in in \}$

Special Case for the last union term, namely Y = Z

```
  ↓ in
*X := Y
  ↓ out
```

$F_{*X := Y}(in) = in - \{ Z \to * \mid Z \in \text{may-point}(X) \} \cup \{ Z \to N \mid Z \in \text{must-point-to}(X) \wedge Y \to N \in in \} \cup \{ Z \to N \mid (Y \to N) \in in \wedge (Z \to N) \in in \}$

```
  ↓ in
*X := *Y + *Z
  ↓ out
```

$F_{*X := *Y + *Z}(in) = F_{a := *Y; b := *Z; c := a + b; *X := c}(in)$

```
  ↓ in
X := G(...)
  ↓ out
```

$F_{X := G(...)}(in) = \emptyset$

How to deal with case of "infinity" ?

Option 2: Map from variables to single lattice

Given lattice $(D, \sqsubseteq_1, \bot_1, \top_1, \sqcup_1, \sqcap_1)$ and a set V, create:

map lattice $V \to D = (V \to D, \sqsubseteq, \bot, \top, \sqcup, \sqcap)$

```
  ↓ in
X := Y op Z
  ↓ out
```

$F_{X := Y \text{ op } Z}(in) = in [X \to in(Y) \, \widehat{op} \, in(Z)]$

where a $\widehat{op}$ b =

$\widehat{op}$	$\bot$	d_1	$\top$
$\bot$	$\bot$	$\bot$	$\top$
d_2	$\bot$	$d_1 \text{ op } d_2$	$\top$
$\top$	$\top$	$\top$	$\top$

Common Sub-expression Elim

```
  ↓ in
X := Y op Z
  ↓ out
```

$F_{X := Y \text{ op } Z}(in) = in - \{ X \to * \} - \{ * \to ... X ... \} \cup \{ X \to Y \text{ op } Z \mid X \neq Y \wedge X \neq Z \}$

```
  ↓ in
X := Y
  ↓ out
```

$F_{X := Y}(in) = in - \{ X \to * \} - \{ * \to ... X ... \} \cup \{ X \to E \mid Y \to E \in in \}$

May and Must

	May	Must
most optimistic (bottom)	empty set	full set
most conservative (top)	full set	empty set
safe	overly big	overly small
merge	∪	∩

Worklist algorithm using lattices

```
let m: map from edge to computed value at edge
let worklist: work list of nodes

for each edge e in CFG do
   m(e) := ⊥

for each node n do
   worklist.add(n)

while (worklist.empty.not) do
   let n := worklist.remove_any;
   let info_in := m(n.incoming_edges);
   let info_out := F(n, info_in);
   for i := 0 .. info_out.length do
      let new_info := m(n.outgoing_edges[i]) ⊔
                          info_out[i];
      if (m(n.outgoing_edges[i]) ≠ new_info])
         m(n.outgoing_edges[i]) := new_info;
         worklist.add(n.outgoing_edges[i].dst);
```

How to calculate Size and Height

$$\text{Size} = 2^{|Vars| \cdot |Z^+|}$$

$$\text{Height} = |Vars| \cdot |Z^+|$$

Cost Function

At the highest level, a cost function should include both a correctness term eq(·) and a performance term, perf(·). An optimization, $\mathcal{R}'$, is any rewrite for which the cost function obtains a minimum value and the correctness term is zero.

$$c(\mathcal{R}; \mathcal{T}) = \text{eq}(\mathcal{R}; \mathcal{T}) + \text{perf}(\mathcal{R}; \mathcal{T}) \qquad (2)$$

$$\mathcal{R}' = \arg\min_{r} \left(\text{perf}(r; \mathcal{T}) \,\middle|\, \text{eq}(r; \mathcal{T}) = 0 \right) \qquad (3)$$

Branch 15 Solution

$F_{\mathbf{branch}(x==c)}(S) = (out_T, out_F)$ where

$$\begin{aligned} in(x) &= \{(x,s) \mid (x,s) \in S\} \\ out_T &= \begin{cases} S - kill(x) \cup (\{(x,+)\} \cap in(x)) & \text{if } c > 0 \\ S - kill(x) \cup (\{(x,0)\} \cap in(x)) & \text{if } c = 0 \\ S - kill(x) \cup (\{(x,-)\} \cap in(x)) & \text{if } c < 0 \end{cases} \\ out_F &= \begin{cases} S - \{(x,0)\} & \text{if } c = 0 \\ S & \text{otherwise} \end{cases} \end{aligned}$$

《Superoptimizer: A Look at the Smallest Program》

1. The paper purposes a tool called Superoptimizer that tries to **find the shortest length assembly language compiled version of a function** (maybe also be refered to as a program).
2. It takes advantage of **convoluted bit shifting features** of some instructions to create smaller compiled function
3. Basically, Superoptimizer's algorithm is to test all possible combinations in the instruction set and find the smallest one that is equivalent to the input. It does this by generating sets of instructions of lenght 1, length 2, etc. until an optimal solution is found
4. **two key routines in Superoptimizer.** The first is quickly testing if the input function and the generated function are equivalent. Superoptimizer utilizes the Probabilistic Test to quickly determine if two functions are equivalent. The other key routine is generating a combination of instructions from the subset to create a possible equivalent function. Clearly, there are some combinations of instructions that can be quickly ruled out. Thus, Superoptimizer prunes the search space of the instruction set by ruling out combinations that cannot be part of the optimal solution
5. In contrast, this version of Superoptimizer can handle functions with 12 instructions and can **test 50,000 programs per second** for equivalence. However, probabilistic test can **lead to incorrect solutions** . Though in the authors experience the probability of finding a **false positive is very low**. Furthermore, the author still has to see a program that is incorrect based on the probabilistic test.
6. Pros and Cons:

Cons: Clearly in terms of limitations, Superoptimizer is still doing an exponential search which causes it only to run currently on functions with 12 instructions or less. Also, pointers present a problem because like addition and subtraction, they have a large set of possible values. And the probabilistic test on them is inconclusive. Also, the instruction subset that the Superoptimizer works with must be able to run on the native machine. Thus, for every new architecture to run Superoptimizer a code change is required.
Pros: First, being helping a compiler with peephole optimizations. Once, Superoptimizer runs and finds optimized solutions to common tasks a compiler deals with. For example, finding a constant offset for an array lookup. Or doing a comparison between two values. The compiler can use the set of optimal generated programs by Superoptimizer to search for sequences in an actual program and patch them in with the optimal solution. A second benefit is for assembly programmer writers that wish to implement stdlib functions as optimally as possible to reduce the space they take up. The author gives the example of optimizing the printf routine in the c stdlib. Finally, analyzing optimal programs can help architecture designers understand what instructions are useful and which ones can be discarded when designing possible instruction sets.

《Stochastic Superoptimization》

1. We formulate the loop-free binary superoptimization task as a **stochastic search problem**. The competing constraints of transformation correctness and performance improvement are encoded as terms in a cost function, and a Markov Chain Monte Carlo sampler is used to rapidly explore the space of all possible programs to find one that is an optimization of a given target program. Although our method sacrifices completeness, the scope of programs we are able to consider, and the resulting quality of the programs that we produce, far exceed those of existing superoptimizers. Beginning from binaries compiled by llvm -O0 for 64-bit x86, our prototype implementation, STOKE, is able to produce programs which either match or outperform the code produced by gcc -O3, icc -O3, and in some cases, expert handwritten assembly.
2. Correctness distance to compute
3. Overcome limitations by : **based on incomplete search.** The competing requirements of correctness and speed are defined as terms **in a cost function over the complex search space** of all loop-free executable hardware instruction sequences, and the program **optimization task** is formulated as a **cost minimization** problem. Employing a **Markov Chain Monte Carlo (MCMC)** sampler to explore the function and produce low-cost samples is sufficient for producing high quality programs. (Minimizing)
4. **Separating Synthesis from Optimization** For loop-free sequences of code, llvm -O0 and gcc -O3 codes differ primarily with respect **to efficient use of the stack and choices of individual instructions**. Except these, the resulting codes are algorithmically quite similar. **Because** optimizers are designed to compose many small local transformations: **dead code elimination deletes one instruction, constant propagation changes one register to an immediate, and strength reduction replaces a multiplication with an add.**
5. Cost minimization into two phases: • Synthesis A synthesis phase focused solely on correctness, which attempts to locate regions of equivalent programs which are distinct from the region occupied by the target. • Optimization An optimization phase focused on speed, which searches for the fastest program within each of those regions. The two phases share the same search implementation; only the starting point and the acceptance functions are different. Synthesis begins with a random starting point (a sequence of randomly chosen instructions), while optimization begins with a code sequence known to be equivalent to the target.