Dataflow Analysis

Fixed Point

If F is monotonic, don't need outer join

- If F is monotonic and height of lattice is finite: iterative algorithm terminates
- If F is monotonic, the fixed point we find is the least fixed point.

Precision Lost

(1) Unreachable code: remove it first

(2) some paths are infeasible

Path merging

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

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

in
`X := *Y`
out

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

因为是 must analysis. 首先删除所有可能 x 指向的，然后加上 x 一定指向的，很有可能删除多了，某些 x 不一定指向的，但是原值和 Y 相等，加回来。

in
`*X := Y`
out

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

Common Sub-expression Elimination (CSE)

in
`X := Y op Z`
out

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

in
`X := Y`
out

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

Liveness

in
`X := Y op Z`
out

$F_{X := Y\ op\ Z}(out) = out - \{ X \} \cup \{ Y, Z \}$

out - {x} ∪

x ∉ out ? ∅ : {Y, Z}

Reaching Definition

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) }

may-point-to

$$
\begin{aligned}
kill(x) &= \bigcup_{v \in Vars} \{(x, v)\} \\
F_{x:=k}(S) &= S - kill(x) \\
F_{x:=a+b}(S) &= S - kill(x) \\
F_{x:=y}(S) &= S - kill(x) \cup \{(x, v) \mid (y, v) \in S\} \\
F_{x:=\&y}(S) &= S - kill(x) \cup \{(x, y)\} \\
F_{x:=*y}(S) &= S - kill(x) \cup \{(x, v) \mid \exists t \in Vars.[(y, t) \in S \wedge (t, v) \in S]\} \\
F_{*x:=y}(S) &= \textbf{let } V := \{v \mid (x, v) \in S\} \textbf{ in} \\
&\quad S - (\textbf{if } V = \{v\} \textbf{ then } kill(v) \textbf{ else } \emptyset) \\
&\quad \cup \{(v, t) \mid v \in V \wedge (y, t) \in S\}
\end{aligned}
$$

Sign

$$
\begin{aligned}
kill(x) &= \bigcup_{s \in \{+,-,0\}} \{(x, s)\} \\
sign(c) &= \begin{cases} + & \text{if } c > 0 \\ - & \text{if } c < 0 \\ 0 & \text{if } c = 0 \end{cases} \\
F_{x:=c}(S) &= S - kill(x) \cup \{(x, sign(c))\} \\
F_{x:=y}(S) &= S - kill(x) \cup \{(x, s) \mid (y, s) \in S\} \\
F_{x:=y\ op\ z}(S) &= S - kill(x) \cup \{(x, s) \mid (y, a) \in S \wedge (z, b) \in S \wedge s = a\ \widehat{op}\ b\}
\end{aligned}
$$

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

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

Power of two

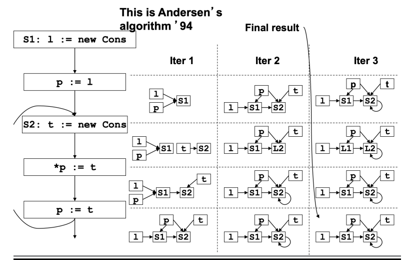
$$F_{X:=C}(S) = S - \{(X, *)\} \cup \{(X, n) \mid C = 2^n\}$$

$$F_{X:=Y*Z}(IN) = IN - \{(X, *)\} \cup \{(X, n+m) \mid (Y, n) \in IN \wedge (Z, m) \in IN\}$$

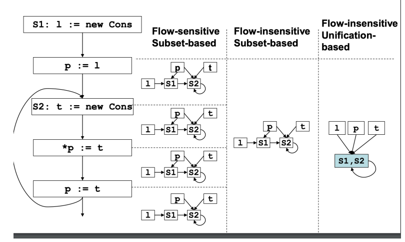
只有当 y 和 z 都是2^n的情况

$$F_{X:=Y+Z}(IN) = IN - \{(X, *)\} \cup \{(X, n+1) \mid (Y, n) \in IN \wedge (Z, n) \in IN\}$$

POINTERS



Steensguard



INTERPROC

Context Insensitivity

```
    main() {
L1:   f()
    }

    f() {
      if(Unlocked()) {
        lock;
L2:     f();
      } else {
        unlock;
      }
    }
```

main ↓	main ↑	f ↓	f ↑
u	∅	∅	∅
"	"	u	"
"	"	u,l	"
"	"	u,l	u
"	u	u,l	u

Context Sensitivity approach 1

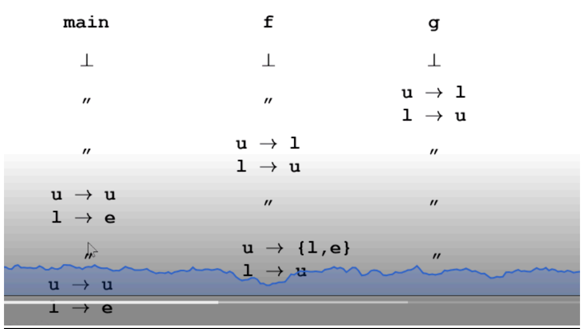
```
L0 main() {          f() {                        g() {
L1   g();          L3  g();          L4              if(isLocked()) {
L2   f();              if (...) { main(); }            unlock;
     lock;           }                               } else { lock; }
     unlock;                                       }
   }
```

main: ↓ ↑
L0 : u ∅
"
"
"
"
L0 u u
L4 u u

f: ↓ ↑
⊥
"
"
L2: l ∅
"
L2: l u

g: ↓ ↑
⊥
L1: u ∅
L1: u l
"
… L1: u l
… L3: l u

approach 2 Context Sensitivity Dataflow based



	Caller precision	Callee precision	code bloat
Inlining	**☺, because contexts are kept separate**	**☺, because contexts are kept separate**	**☹ may be large if we want to get the best precision**
context-insensitive interproc	**😐, because contexts are merged**	**😐, because contexts are merged**	**☺ none**
Context sensitive interproc	**☺, because of context sensitive summaries**	**😐, because contexts are still merged when optimizing callees**	**☺ none**
Specialization	**☺, contexts are kept separate**	**☺, contexts are kept separate**	**😐 Some, less than inlining**

	May	Must
most optimistic (bottom)	empty set	full set
most conservative (top)	full set	empty set
safe	overly big	overly small
merge	$\cup$	$\cap$

<u><Provably Correct Peephole Optimizations with Alive》</u>

1. This paper presents Alive, a domain-specific language for writing optimizations and for automatically either proving them correct or else generating counterexamples.
2. Alive's most important features include its abstraction over choice of constants, over the bitwidths of operands and over LLVM's instruction attributes that control undefined behavior
3. **Correctness Criteria**(1) Target invokes undefined behavior only when the source does(2)Result of target = result of source when source does not invoke undefined behavior (3) Final memory states are equivalent **Same behavior**
4. **3 undefined in LLVM** Poison values, Undef values, True UB
5. Alive transformations are parametric over types. Hence, Alive must verify a transformation for all valid type assignments.

<u>《 End-to-end Deep Learning of Optimization Heuristics》</u>

1. develop a deep neural network that learns heuristics over raw code.
2. with deep neural networks we can **bypass static feature extraction** and learn optimization heuristics directly on raw code
3. Our **system admits auxiliary features to** describe information unavailable at compile time, such as the sizes of runtime input parameters
4. **transfer learning** The properties of the raw code that are **abstracted by the beginning layers of our neural networks** are mostly independent of the optimization problem.
5. **evaluated** heterogeneous device mapping and GPU thread coarsening. **predicting the optimal device** to run a given program, and **predicting thread coarsening factors.**
6. Architecture: LSTM + Auxiliary Input + NN
7. Effective representation should be: **derive** semantic and syntactic patterns of a programming language entirely from sample codes; **identify** the patterns and representation in source codes which are relevant to the task at hand; **discriminate** performance characteristics arising from potentially subtle differences in similar codes.
7. Source Rewriter: LLVM pass, parse the AST, removing conditional compilation **rebuild the input source code using a consistent code style and identifier naming scheme**
8. **Sigmoid The activation** of each neuron in the output layer represents the model's confidence that the corresponding decision is the correct one. We take the **arg max** of the output layer to find the decision with the largest activation.