Formalization of DFA using lattices

Recall worklist algorithm

\[
\begin{align*}
\text{let } m &: \text{ map from edge to computed value at edge} \\
\text{let } \text{worklist}: \text{ work list of nodes} \\
\text{for each edge } e \text{ in CFG do} \\
\quad m(e) := \emptyset \\
\text{for each node } n \text{ do} \\
\quad \text{worklist.add}(n) \\
\text{while } (\text{worklist.empty}.\text{not}) \text{ do} \\
\quad \text{let } n := \text{worklist.remove.any} \\
\quad \text{let } \text{info}_\text{in} := m(n.\text{incoming_edges}) \\
\quad \text{let } \text{info}_\text{out} := F(n, \text{info}_\text{in}) \\
\quad \text{for } i = 0 \text{ to } \text{info}_\text{out}.\text{length} \text{ do} \\
\quad \quad \text{let } \text{new}_\text{info} := m(n.\text{outgoing_edges}[i]) \cup \text{info}_\text{out}[i] \\
\quad \quad \text{if } m(n.\text{outgoing_edges}[i]) \neq \text{new}_\text{info} \\
\quad \quad \quad m(n.\text{outgoing_edges}[i]) := \text{new}_\text{info} \\
\quad \text{worklist.add}(n.\text{outgoing_edges}[i].\text{dst}) \\
\end{align*}
\]

Using lattices

- We formalize our domain with a powerset lattice
- What should be top and what should be bottom?

Using lattices

- We formalize our domain with a powerset lattice
- What should be top and what should be bottom?
- Does it matter?
  - It matters because, as we’ve seen, there is a notion of approximation, and this notion shows up in the lattice

Using lattices

- Unfortunately:
  - dataflow analysis community has picked one direction
  - abstract interpretation community has picked the other
- We will work with the abstract interpretation direction
- Bottom is the most precise (optimistic) answer, Top the most imprecise (conservative)

Direction of lattice

- Always safe to go up in the lattice
- Can always set the result to \( \top \)
- Hard to go down in the lattice
- Bottom will be the empty set in reaching defs
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) := \perp

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]) \neq new_info)
            m(n.outgoing_edges[i]) := new_info;
        worklist.add(n.outgoing_edges[i].dst);

Termination of this algorithm?

• For reaching definitions, it terminates...
• Why?
  – lattice is finite
• Can we loosen this requirement?
  – Yes, we only require the lattice to have a finite height
• Height of a lattice: length of the longest ascending or descending chain
• Height of lattice \((2^S, \subseteq) = \)

Even more formal

• To reason more formally about termination and precision, we re-express our worklist algorithm mathematically

• We will use fixed points to formalize our algorithm

Termination

• Still, it’s annoying to have to perform a join in the worklist algorithm

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]) \cup info_out[i];
        if (m(n.outgoing_edges[i]) \neq new_info)
            m(n.outgoing_edges[i]) := new_info;
        worklist.add(n.outgoing_edges[i].dst);

Fixed points

• Recall, we are computing m, a map from edges to dataflow information
• Define a global flow function F as follows: F takes a map m as a parameter and returns a new map m’, in which individual local flow functions have been applied
Fixed points

- We want to find a fixed point of $F$, that is to say a map $m$ such that $m = F(m)$
- Approach to doing this?
  - Define $\bot$, which is $\bot$ lifted to be a map: $\bot = \lambda e. \bot$
  - Compute $F(\bot)$, then $F(F(\bot))$, then $F(F(F(\bot)))$, ... until the result doesn’t change anymore

- Formally:

  $\exists_{\text{monotonic}}$
  $
  \text{Outer join has same role here as in worklist algorithm: guarantee that results keep increasing}
  \text{BUT: if the sequence } F^i(\bot) \text{ for } i = 0, 1, 2 ... \text{ is increasing, we can get rid of the outer join!}
  \text{How? Require that } F \text{ be monotonic:}
  \quad \forall a, b . a \sqsubseteq b \Rightarrow F(a) \sqsubseteq F(b)

- Another benefit of monotonicity
  - Suppose Marsians came to earth, and miraculously give you a fixed point of $F$, call it $fp$
  - Then:
Another benefit of monotonicity

• Suppose Marsians came to earth, and miraculously give you a fixed point of $F$, call it $fp$.

Then:

\[ \neg \exists i \in I \quad \wedge \quad F(i) \subseteq F(fp) \]

\[ F(I) \subseteq fp \]

\[ F^{-1}(I) \subseteq fp \]

\[ \sigma_{fp} \subseteq I \]

Another benefit of monotonicity

• We are computing the least fixed point...

Recap

• Let’s do a recap of what we’ve seen so far

• Started with worklist algorithm for reaching definitions

Worklist algorithm for reaching defns

```plaintext
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]) \[\text{if } (m(n.outgoing_edges[i]) \neq \text{new_info})\]
        m(n.outgoing_edges[i]) := new_info;
        worklist.add(n.outgoing_edges[i].dst);
```

Generalized algorithm using lattices

```plaintext
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]) \[\text{if } (m(n.outgoing_edges[i]) \neq \text{new_info})\]
        m(n.outgoing_edges[i]) := new_info;
        worklist.add(n.outgoing_edges[i].dst);
```

Next step: removed outer join

• Wanted to remove the outer join, while still providing termination guarantee

• To do this, we re-expressed our algorithm more formally

• We first defined a "global" flow function $F$, and then expressed our algorithm as a fixed point computation
Guarantees

- 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.

What about if we start at top?

- What if we start with $\top: F(\top), F(F(\top)), F(F(F(\top)))$?
- We get the greatest fixed point
- Why do we prefer the least fixed point?
  - More precise

Graphically

[Graph showing the iterative process and fixed points]
Graphically, another way