Solving Data Flow Frameworks

Frameworks abstract from specific analysis problems

- leverage theoretical properties about frameworks for an analysis
- generic solution algorithms for frameworks

MFP (Maximal Fixed Point) solution

- computes the *least* solution to the equation system induced by the framework

Misnomer due to the fact that for classic problems

- where $\sqcup$ is $\cap$
- one can consider the least solution for superset ordering, $\sqsubseteq = \supseteq$,
- or equivalently the greatest solution for subset ordering, $\sqsubseteq = \subseteq$.

Recall: A Framework Instance

An instance consists of

- A framework, i.e., L and $\mathcal{F}$
- a finite flow relation, F (e.g., $flow(S_*)$)
- a finite set of extremal labels, E (e.g., $final(S_*)$)
- an extremal value, $\iota \in L$
- a mapping, $f_\cdot$, from the labels Lab_*, mentioned in F and E, to functions in $\mathcal{F}$

A Framework Instance Defines a System of Equations

$$\text{Analysis}_\circ(l) = \bigsqcup\{\text{Analysis}_\bullet(l') | (l', l) \in F\} \sqcup \iota_E^l$$

$$\text{where } \iota_E^l = \begin{cases} \iota & \text{if } l \in E \\ \perp & \text{if } l \notin E \end{cases} \quad \text{and}$$

$$\text{Analysis}_\bullet(l) = f_l(\text{Analysis}_\circ(l))$$

Iterative Worklist Algorithm for Solving Data Flow Frameworks

INPUT: An Instance of a Monotone Framework $(L, \mathcal{F}, F, E, \iota, f.)$

OUTPUT: $MFP_\circ, MFP_\bullet$

ALGORITHM: Using two data structures

`W`, a list of edges, and

`Analysis`, a mapping from labels to values in L

Step 1: Initialize `W` and `Analysis`

Step 2: Iterate updating values of `W` and `Analysis` using $f.$ until no new values are produced

Step 3: Collect results from `Analysis`

Iterative Worklist Algorithm : Step 1

Step 1: Initialize `W` and `Analysis`

[1] `W := nil;`

[2] `forall` $(l, l') \in F$ `do`

[3]   `W := cons(`(l, l')`,W);`

[4] `orall` l `appearing in` F `or in` E `do`

[5]   `if` $l \in E$ `then`

[6]   `Analysis[`l`] :=` ι

        `else`

[7]   `Analysis[`l`] :=` $\perp_L$`;`

Iterative Worklist Algorithm : Step 2

Step 2: Iterate updating values of `W` and `Analysis` using f. until no new values are produced

`while W` $\neq$ `nil do`
  l `:= first(head(W));`
  l' `:= second(head(W));`
  `W := tail(W);`
  `if` f_l`( Analysis[`l`])` $\not\sqsubseteq$ `Analysis[`l'`] then`
    `Analysis[`l'`] := Analysis[`l'`]` $\sqcup$ f_l`(Analysis[`l`]);`
    `forall` l'' `with` $(l', l'') \in F$ `do`
      `W := cons(`(l', l'')`,W);`

Iterative Worklist Algorithm : Step 3

Step 3: Collect results from `Analysis`

`forall` l `appearing in` F `or in` E `do`

    $MFP_{\circ}(l)$ `:= Analysis[`l`];`

    $MFP_{\bullet}(l)$ `:=` f_l`(Analysis[`l`])`

Properties of Algorithm

Termination

Correctness

Run-time bound

Algorithm Terminates

Steps 1 and 3 iterate over the extremal and edge sets (which are finite for any flow graph)

For step 2 assume that there are b labels in the program and that the flow graph is fully connected

- `W` initially has at most b^2 edges

Algorithm Terminates

Each outer iteration either deletes an edge (lines `9−11`) or adds up to b edges (line `15`).

Edges are only added if

- new information is produced on output
- i.e., $f_l(\texttt{Analysis}[l]) \sqsupset \texttt{Analysis}[l']$
- or the values are incomparable in the lattice

The result is that the successor's input, l', is $\sqsupset$ than l's output, but this can only happen finitely many times

- since ascending chain condition, $l_1 \sqsubseteq l_2 \sqsubseteq \ldots$, assure that $\exists n : l_n = l_{n+1} = \ldots$

Algorithm Computes MFP

Let $Analysis_\circ$ and $Analysis_\bullet$ be the least solution to the given framework instance

Two parts to the proof

1. Show that `Analysis` values approximate Analysis$_\circ$ values on each iteration in step 2.
2. Show that Analysis$_\circ$ values approximate `Analysis` values at termination of step 2.

Algorithm Computes MFP : Part 1

Want to show that $\forall l : \texttt{Analysis}[l] \sqsubseteq \mathit{Analysis}_\circ(l)$ is a loop invariant for step 2

After step 1 the invariant holds since either

- $\texttt{Analysis}[l] = \iota \sqsubseteq \text{Analysis}_\circ(l)$
- $\texttt{Analysis}[l] = \bot_L \sqsubseteq \text{Analysis}_\circ(l)$

In step 2, either

- no values change (when an edge is deleted)
- a single value changes (the value for the source of the edge)

Algorithm Computes MFP : Part 1

Let $(l, l') \in F$ be the one removed from the worklist in lines $\texttt{9-11}$

Then assuming the invariant held at the start of this iteration

- *new*$\texttt{Analysis}[l'] = \texttt{Analysis}[l'] \sqcup f_l(\texttt{Analysis}[l])$ (line $\texttt{13}$)
- *new*$\texttt{Analysis}[l'] \sqsubseteq \text{Analysis}_\circ[l'] \sqcup f_l(\text{Analysis}_\circ[l])$ (from invariant and monotonicity)
- *new*$\texttt{Analysis}[l'] \sqsubseteq \text{Analysis}_\circ[l']$ (since RHS is the least fixed point $\sqcup$ adds nothing)

So, the invariant holds.

Algorithm Computes MFP : Part 2

Loop in step 2 terminates when `W` is empty

We show that the result leaves the input of a successor at least as high in the lattice as the output of its predecessor

- $\forall l, l' : (l, l') \in F \Rightarrow \texttt{Analysis}[l'] \sqsupseteq f_l(\texttt{Analysis}[l])$
- the successor input may be higher due to another of its predecessors

by contradiction.

Algorithm Computes MFP : Part 2

Suppose $\exists l, l' : (l, l') \in F \Rightarrow \mathtt{Analysis}[l'] \not\sqsupseteq f_l(\mathtt{Analysis}[l])$

If $\mathtt{Analysis}[l]$ was last updated in step 1 (e.g., because l is the entry), then (l, l') was processed in step 2 and

- $\mathtt{Analysis}[l'] \sqsupset f_l(\mathtt{Analysis}[l])$ holds because of failure to pass test at line 12
- $\mathtt{Analysis}[l'] \sqsupseteq f_l(\mathtt{Analysis}[l])$ holds because assignment at line 12

This property is maintained for each subsequent processing of (l, l')

If $\mathtt{Analysis}[l]$ last updated in step 2, then (l, l') was put on W at line 15 and the same reasoning applies.

Thus, no such (l, l') can exist.

Algorithm Computes MFP : Part 2

On termination of loop in step 2

$$\forall l \in E : \texttt{Analysis}[l] \sqsupseteq \iota$$

holds, since it was established at line 6 and there are no incoming flows to extremal points.

Thus on termination of the loop in step 2

- Analysis values on input to a node are higher than all predecessor output values and the initial value for that node
- $\forall l : \texttt{Analysis}[l'] \sqsupseteq (\bigsqcup\{f_{l'}(\texttt{Analysis}[l']) \mid (l', l) \in F\}) \sqcup \iota_E^l$

Algorithm Computes MFP

From part 1 we have

$$\forall l : \texttt{Analysis}[l] \sqsubseteq \text{Analysis}_\circ(l)$$

From part 2 we have

$$\forall l : \texttt{Analysis}[l] \sqsupseteq \text{Analysis}_\circ(l)$$

since the l's value of the RHS can be shown to be the $\bigsqcup$ of all the $f_{l'}$ for all predecessors l'

In step 3 of the algorithm $\text{MFP}_\circ$ is defined such that

$$\forall l : \text{MFP}_\circ(l) = \texttt{Analysis}[l]$$

So $\text{MFP}_\circ = \text{Analysis}_\circ(l)$.

Algorithm Run-time Bound

Let $b \geq 1$ be the number of labels, $e \geq b$ be the number of edges, and $h \geq 1$ be the (finite) height of the lattice.

Step 1 and 3 are $\mathcal{O}(b + e)$ operations

- iterate over elements of F and E

In step 2, an edge is added at most $\mathcal{O}(h)$ times to `W`

- minimal change in value of l' means that we can have at most h such moves (from $\perp_L$ to $\top_L$)

each such addition requires a constant number of operations except for the list insertion, which is $\mathcal{O}(e)$.

Since $h \geq 1$ and $e \geq b$, $\mathcal{O}(b + e)$ is bounded by $\mathcal{O}(e \cdot h)$

Often times you'll see the bound expressed as $\mathcal{O}(b^2 \cdot h)$

Meet Over Paths (MOP) Solution

We really are interested in information corresponding to program executions

- executions modeled as paths through flow graph
- MFP solution approximates path information by combining information on a per edge basis
- may lose information

Let $(L, \mathcal{F}, F, E, \iota, f.)$ be a framework.

Meet Over Paths (MOP) Solution

Denote with $\vec{l} = [l_1, \ldots, l_n]$ be a sequence of labels.

The paths leading up to but not including l are

$$\begin{array}{rcl} path_\circ(l) & = & \{[l_1, \ldots, l_{n-1}] \mid n \geq 1 \wedge \\ & & \forall i < n : (l_i, l_{i+1}) \in F \wedge l_n = l \wedge l_1 \in E\} \end{array}$$

The paths leading up to and including l are

$$\begin{array}{rcl} path_\circ(l) & = & \{[l_1, \ldots, l_n] \mid n \geq 1 \wedge \\ & & \forall i < n : (l_i, l_{i+1}) \in F \wedge l_n = l \wedge l_1 \in E\} \end{array}$$

Meet Over Paths (MOP) Solution

For a path $\vec{l} = [l_1, \ldots, l_n]$ define its transfer function as

$$f_{\vec{l}} = f_{l_n} \circ \ldots \circ f_{l_1} \circ id$$

where id is the identity function.

Now we define meet over paths as

$$\text{MOP}_\circ(l) = \bigsqcup \{ f_{\vec{l}}(\iota) \mid \vec{l} \in path_\circ(l) \}$$

MOP is Undecidable for Some Problems

A *correspondance system* is a set of pairs, $P : \Sigma^+ \times \Sigma^+$, where the components are non-empty strings drawn from the same alphabet.

A *modified* correspondance system fixes one of the pairs, e.g., $u_i = v_i$.

A *match* of a correspondance system is a string $w \in \Sigma^+$ such that, for some $n > 0$ and a set of pairs $(u_1, v_1), (u_2, v_2), \ldots, (u_n, v_n) \in P$

$$w = u_1 u_2 \ldots u_n = v_1 v_2 \ldots v_n$$

The determination of a match for a given correspondance system (modified or not) is undecidable (*Post's correspondance problem*).

MOP for Constant Propogation is Undecidable

To show that the MOP solution for constant propogation is undecidable

- we need to find a program that encodes a correspondance problem
- the MOP solution having a specific value determines a match

Let $\Sigma = \{1, \ldots, 9\}$, $\mid x \mid$ be the length of string x, and $[\![x]\!]$ be the numeric value of string x.

Consider the modified correspondance problem where $u_1 = v_1 = 1$; we write numbers and their string representations identically.

MOP for Constant Propogation is Undecidable

```
x := 1; y := 1;
while [...] do (
  if [...] then
    x := x*10^|u_1| + [[u_1]];
    y := y*10^|v_1| + [[v_1]]
  else if [...] then
    ...
  if [...] then
    x := x*10^|u_n| + [[u_n]];
    y := y*10^|v_n| + [[v_n]]
)
z := sign((x-y)*(x-y));
```

If the MOP solution of constant propogation determines that $z = 1$ then there is no solution to the modified Post correspondance problem.

MOP for Constant Propogation is Undecidable

To illustrate the idea, consider the case where

- $n = 2, w = 123, u_1 = 1, u_2 = 23, v_1 = 12, v_2 = 3$

A MOP solution considers each program path separately and then meets the results.

Ignoring branch conditions, there is a path going through the first and second conditional

- $x = 1; y = 1$
- $x = 1 \cdot 10^1 + 1 = 11; y = 1 \cdot 10^2 + 12 = 112$
- $x = 11 \cdot 10^2 + 23 = 1123; y = 112 \cdot 10^1 + 3 = 1123$

This will yield $z = 0$ and this corresponds to a problem with a solution.

Section 2.4

Details the relationship between MFP and MOP solutions

For Monotone Frameworks, $MFP_\circ \sqsupseteq MOP_\circ$

- The MOP solution may be more precise (if you can calculate it)

For Distributive Frameworks, $MFP_\circ = MOP_\circ$

- the classic problems (LV, AE, VBE, RD) were distributive
- all *Gen-Kill* (or *Bit Vector*) frameworks are distributive

Improving the Solution Algorithm

In our discussion we have hinted that there may be some ways to improve the basic algorithm

- do not allow duplicates in the worklist
- visit nodes in a *good* sequence

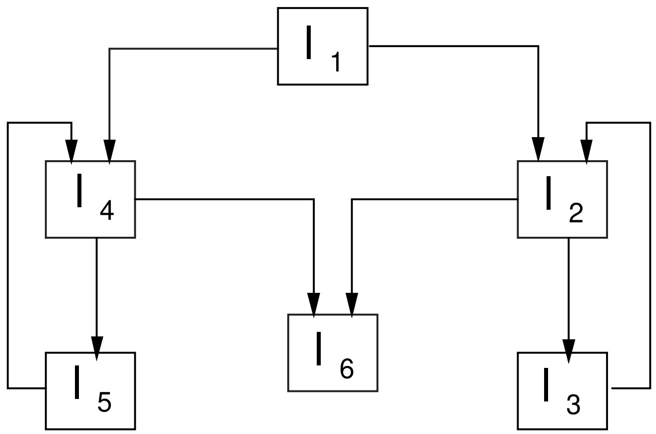
PPA Chapter 6 outlines some of these optimizations

An Example

Consider the following program

if $[b_1]^1$ then
    while $[b_2]^2$ do $[x := a_1]^3$
else
    while $[b_3]^4$ do $[x := a_2]^5$;
$[x := a_3]^6$

whose flow graph looks like this



Worklist Node Ordering

FIFO ordering will perform a depth first traversal of the graph

LIFO ordering will perform a breadth first traversal of the graph

Its easy to find examples where either of these will result in lots of node visitations that fail the test at line 12.

We want to visit nodes in execution order (for forward problems) and in reverse execution order (for backward problems)

Reverse Post-Order

You know about the basic kinds of tree traversals

- pre-order, e.g., "+ x y"
- in-order, e.g., "x + y"
- post-order, e.g., "x y +"

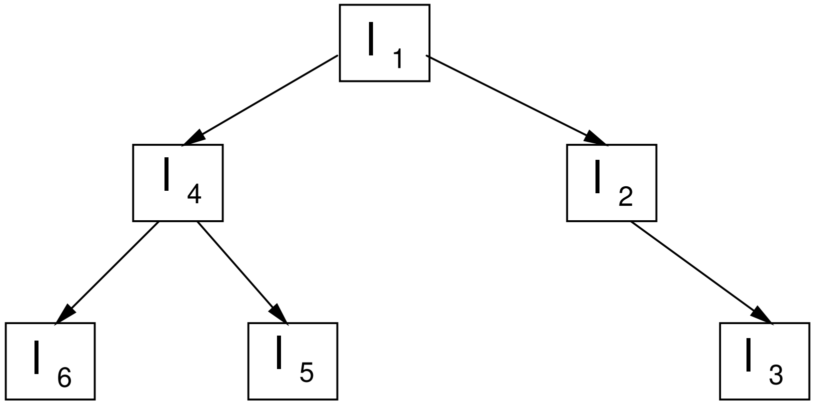
We calculate the *depth first spanning forest* for the flow graph

- this yields a set of trees
- that can be traversed in sequence
- performing a post-order traversal assigning numbers starting at $\mid \mathrm{Lab}_* \mid$ down to 1

This yields a *reverse post-order numbering*

An Example

The depth-first spanning forest (DFSF) (in this case tree) looks like



This yields the order: $l_1, \ldots, l_6$

Note that l_5 and l_6 could be swapped in the tree/order without a problem.

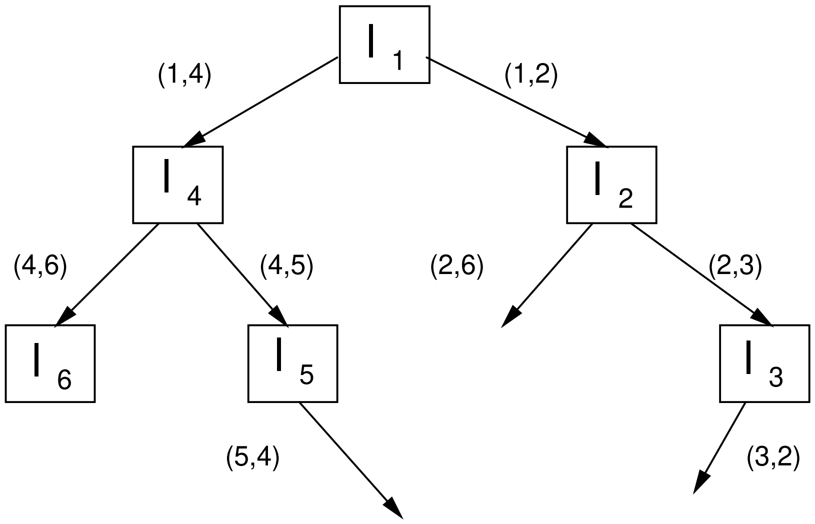
This order insures that paths are visited in sequence

- e.g., l_1, l_4, l_5
- e.g., l_1, l_2, l_3

rPostOrder for Edge-oriented Algorithms

Our algorithm operates on edges, not nodes of the flow graph, thus we will want an ordering on edges.

It is easy to extend the DSFS algorithm to produce a reverse post order numbering for edges.



This yields the order:

$$(1,2),(2,6),(2,3),(3,2),(1,4),(4,6),(4,5),(5,4)$$

Round Robin Algorithm for Solving Data Flow Frameworks

INPUT: Framework Instance $(L, \mathcal{F}, F, E, \iota, f.)$

OUTPUT: $MFP_\circ, MFP_\bullet$

ALGORITHM: *no worklist!*

Step 1: Initialize `Analysis`

forall l appearing in F or in E do
    Analysis[l] := ι_E^l
change := true;

Step 2: Iterate updating values of `Analysis`

while change do
  change := false;
  forall $(l, l') \in F$ in reverse post order
    if f_l(Analysis[l]) $\not\sqsubseteq$ Analysis[l'] then
      Analysis[l'] := Analysis[l'] $\sqcup$ f_l(Analysis[l]);
      change := true;

Is this an Improvement?

In practice, yes, but for general frameworks the bound is unchanged.

Exploiting properties of the functions in Gen-Kill frameworks and the fact that while programs are structured one gets the following results

> For Gen-Kill Frameworks the round robin algorithm terminates in $\mathcal{O}((d+1)\cdot b)$, where b is the number of labelled statements and d is the maximum depth of loop nesting.

In general d is small (less than 5), so this is a big win compared to b.

Reverse Post Order

Pushes data flow information along paths

Convergence is forestalled by the presence of loops so why don't we try to process loops specially

- iterate cycles until they converge locally
- then propogate inter-loop information
- repeat until no more changes

Section 6.3 describes this approach in some detail.

PAG

Supports several orderings

- depth-first (FIFO)
- breadth-first (LIFO)
- strongly connected components (SCC)
- several variations with SCC approaches

No rPostOrder

but the SCC approaches do essentially the same thing

Try some experiments by varying the order on some while programs