DFA of non-distributive properties
The general pattern of Dataflow Analysis

\[
\begin{align*}
GA(p) = \\
\begin{cases}
\iota & \text{if } p \in E \\
\oplus \{ GA(q) \mid q \in F \} & \text{otherwise}
\end{cases}
\end{align*}
\]

\[GA(p) = f_p(GA(p))\]

where:
- \(E\) is the set of initial/final points of the control-flow diagram
- \(\iota\) specifies the initial values
- \(F\) is the set of successor/predecessor points
- \(\oplus\) is the combination operator
- \(f\) is the transfer function associated to node \(p\)
Distributive properties

- Monotonicity of a function implies that

\[ f(x \cup y) \supseteq f(x) \cup f(y) \]

- A function is said **distributive** if a stronger condition holds:

\[ f(x \cup y) = f(x) \cup f(y) \]

- In general, a dataflow analysis is said distributive if the transfer functions satisfy

\[ f(\text{lub}(x, y)) = \text{lub}(f(x), f(y)) \]
Example: if distributive
Example: f not distributive
Why distributivity is important

\[ k(h(f(0) \cup g(0))) = \]
\[ k(h(f(0)) \cup h(g(0)))) = \]
\[ k(h(f(0))) \cup k(h(g(0)))) \]

The overall analysis is equal to the lub of the analyses on the different paths.
DFA of a distributive property

• If the property is distributive, then the minimal solution of the equation system is equivalent to combining the result of the analyses along all the paths (including infinite paths).

• In this case the combination operator (least upper bound) does not introduce further loss of accuracy
Which properties are distributive?

- The distributive properties are usually “easy”
- They mainly concern the structure of the program (not the actual values assigned to the variables)
  - E.g., live variables, available expressions, reaching definitions, very busy expressions
  - These properties concern HOW the program pursues the computation, not the actual values of the variables
Non-distributive properties

• They deal with WHAT a program computes
  – E.g.: has the output always the same constant value? Is a variable always assigned a positive number?

• Example: *Constant Propagation Analysis*

  For each program point, we want to know if a variable is always assigned to exactly the same constant value.

  It is a *forward and definite property*. 
Constant Propagation Analysis

Consider the set: \((\text{Var} \rightarrow \mathbb{Z}^T)_\perp\)

- \textbf{Var} is the set of variables occurring in the program
- \(\mathbb{Z}^T = \mathbb{Z} \cup \{T\}\) partially ordered by:
  \[
  \forall n \in \mathbb{Z} : \quad n \leq_{CP} T \\
  \forall n_1, n_2 \in \mathbb{Z} : \quad (n_1 \leq_{CP} n_2) \iff (n_1 = n_2)
  \]
$\mathbb{Z}^T$

$\mathbb{Z} \
\{T\}

\forall n \in \mathbb{Z} : n \leq T$
The lattice \((\text{Var} \rightarrow \mathbb{Z}^T) \perp\)

- In \(\mathbb{Z}^T\), the top element \(T\) says that a variable is not always assigned to the same constant value (i.e. it may be assigned to different values).

- An element \(\sigma: \text{Var} \rightarrow \mathbb{Z}^T\) is a partial function given a variable \(x\), \(\sigma(x)\) tells us if \(x\) is a constant or not, and in the positive case (if \(\sigma(x)\) is different from \(T\)) what is its value.

- The bottom element \(\perp\) is added to complete the lattice.
The order in \((\text{Var} \rightarrow \mathbb{Z}^\top)_\perp\)

- A partial order in \((\text{Var} \rightarrow \mathbb{Z}^\top)_\perp\)

\[
\forall \sigma \in (\text{Var} \rightarrow \mathbb{Z}^\top)_\perp : \quad \bot \leq \sigma \\
\forall \sigma_1, \sigma_2 \in (\text{Var} \rightarrow \mathbb{Z}^\top)_\perp : (\sigma_1 \leq \sigma_2) \iff (\forall x \in \text{dom}(\sigma_1) : \sigma_1(x) \leq_{\text{CP}} \sigma_2(x))
\]

- The **least upper bound**:

\[
\forall \sigma \in (\text{Var} \rightarrow \mathbb{Z}^\top)_\perp : \text{lub}(\bot, \sigma) = \text{lub}(\sigma, \bot) = \sigma \\
\forall \sigma_1, \sigma_2 \in (\text{Var} \rightarrow \mathbb{Z}^\top)_\perp \\
\forall x \in \text{Var} : \text{lub}(\sigma_1, \sigma_2)(x) = \text{lub}(\sigma_1(x), \sigma_2(x))
\]

Means equality when \(\sigma_i(x)\) are in \(\mathbb{Z}\)!
\( (\{x,y\} \rightarrow \mathbb{Z}^T)_\perp \)
Expression evaluation

- In order to specify the transfer functions, we have to evaluate an expression given a state \( \sigma \) in \((\text{Var} \rightarrow \mathbb{Z}^T)_\perp\)

\[
\mathcal{A}: (\text{AExp} \cdot (\text{Var} \rightarrow \mathbb{Z}^T)_\perp) \rightarrow \mathbb{Z}^T_{\perp}
\]

\[
\mathcal{A}(x, \sigma) = \perp \quad \text{if } \sigma = \perp
\]
\[
\sigma(x) \quad \text{otherwise}
\]

\[
\mathcal{A}(n, \sigma) = \perp \quad \text{if } \sigma = \perp
\]
\[
n \quad \text{otherwise}
\]

\[
\mathcal{A}(a_1 \ \text{op} \ a_2, \sigma) = \mathcal{A}(a_1, \sigma) \ \text{op} \ \mathcal{A}(a_2, \sigma)
\]

(where \text{op} is the corresponding operation of \text{op} on \(\mathbb{Z}^T_{\perp}\): e.g. \(4 \ \text{op} \ 2 = 6\))
Transfer functions

• For Constant Propagation Analysis the set of transfer functions is a subset of

\[ \mathcal{F} = \{ f : (\text{Var} \to \mathbb{Z}^T)_\perp \to (\text{Var} \to \mathbb{Z}^T)_\perp | \ f \text{ monotone} \} \]

• The transfer functions \( f_\ell \) are defined by:

  if \( \ell \) is the label of an assignment \([x := a]^{\ell}\)

  \[
  f_\ell(\sigma) = \begin{cases} 
  \perp & \text{if } \sigma = \perp \\
  \sigma[x \to \mathcal{A}(a,\sigma)] & \text{otherwise}
  \end{cases}
  \]

  if \( \ell \) is the label of another statement: \( f_\ell(\sigma) = \sigma \)
Example

- \([x:=10]^{1}; [y:=x+10]^{2}; ([\text{while } x<y]^{3} [y:=y-1]^{4}); [z:=x-1]^{5}\)

- The minimal solution of the Constant Propagation Analysis of this program is:

- \(\text{CP}_{\text{entry}}(1) = \emptyset\)
- \(\text{CP}_{\text{exit}}(1) = \{(x \rightarrow 10)\}\)
- \(\text{CP}_{\text{entry}}(2) = \{(x \rightarrow 10)\}\)
- \(\text{CP}_{\text{exit}}(2) = \{(x \rightarrow 10), (y \rightarrow 20)\}\)
- \(\text{CP}_{\text{entry}}(3) = \text{CP}_{\text{exit}}(3) = \text{CP}_{\text{entry}}(4) = \text{CP}_{\text{exit}}(4) = \{(x \rightarrow 10), (y \rightarrow \text{T})\}\)
- \(\text{CP}_{\text{entry}}(5) = \{(x \rightarrow 10), (y \rightarrow \text{T})\}\)
- \(\text{CP}_{\text{exit}}(5) = \{(x \rightarrow 10), (y \rightarrow \text{T}), (z \rightarrow 9)\}\)
Non-distributivity

- In order to show that Constant Propagation Analysis is non-distributive, just consider the transfer function $f_\ell$ corresponding to the statement $[y := x \cdot x]_\ell$

  consider two states $\sigma_1(x) = 1$ and $\sigma_2(x) = -1$

  in this case:

  $$\text{lub}(\sigma_1, \sigma_2)(x) = T$$

  and then

  $$f_\ell (\text{lub}(\sigma_1, \sigma_2))(y) = T$$

  whereas

  $$f_\ell (\sigma_1)(y) = 1 = f_\ell (\sigma_2)(y)$$
Interprocedural analysis
Interprocedural Optimizations

– Until now, we have only considered optimizations “within a procedure”
– Extending these approaches outside of the procedural space involves similar techniques:
  • Performing interprocedural analysis
    – Control flow
    – Data flow
  • Using that information to perform interprocedural optimizations
What makes this difficult?

Use worst case assumptions about side effects…
leads to imprecise inaprocedural information
leads to explosion in inaprocedural def-use chains

What happens at a procedure call?

procedure joe(i,j,k)
  l ← 2 * k
  if (j = 100)
    then m ← 10 * j
    else m ← i
  call ralph(l,m,k)
o ← m * 2
q ← 2
call ralph(o,q,k)
write q, m, o, l

procedure main
  call joe(10, 100, 1000)

procedure ralph(a,b,c)
b ← a * c / 2000

What value is printed for q?
Did ralph() change it?

Since j = 100 this always executes the then clause
and always m has the value 1000
What makes this difficult?

What happens at a procedure call?

- Use worst case assumptions about side effects
- Leads to imprecise _intraprocedural_ information
- Leads to explosion in _intraprocedural_ def-use chains

```
procedure joe(i,j,k)
    l ← 2 * k
    if (j = 100)
        then m ← 10 * j
        else m ← i
    call ralph(l,m,k)
    o ← m * 2
    q ← 2
    call ralph(o,q,k)
    write q, m, o, l

procedure main
    call joe( 10, 100, 1000)

procedure ralph(a,b,c)
    b ← a * c / 2000
```

What value is printed for q?
Did ralph() change it?

Since j = 100 this always executes the then clause
and always m has the value 1000

With perfect knowledge, the compiler could replace this with
```
write 2, 1000, 2000, 2000
and the rest is dead!
```

What value is printed for q?
Did ralph() change it?
The general pattern of Dataflow Analysis

\[ GA_\circ(p) = \begin{cases} \top & \text{if } p \in E \\ \bigoplus \{ GA_\circ(q) \mid q \in F \} & \text{otherwise} \end{cases} \]

\[ GA_\circ(p) = f_p \left( GA_\circ(p) \right) \]

where:
- \( E \) is the set of initial/final points of the control-flow diagram
- \( \top \) specifies the initial values
- \( F \) is the set of successor/predecessor points
- \( \bigoplus \) is the combination operator
- \( f \) is the transfer function associated to node \( p \)
Procedure calls

- We can label a procedure call by:

\[\text{[call } p(a,z)]^{\ell_c}_{\ell_r}\]

dove:
- a is an input parameter
- z is an output parameter
- \(\ell_c\) is a label corresponding to the entrance into p
- \(\ell_r\) is a label corresponding to the exit out of p
Flow

- In the intraprocedural analysis we considered a flow as a set of pairs (p,q) corresponding to an edge in the control flow graph.

- We can now consider the call \([\text{call } p(a,z)]^{\ell_c}_{\ell_r}\) and a procedure declaration \(\text{proc } p(\text{val } x, \text{res } y) \text{ is }^{\ell_{\text{in}}} S \text{ end}^{\ell_{\text{out}}};\)

- In the interprocedural graph we should then consider also:
  - \((\ell_c; \ell_{\text{in}})\) the flow from the call \(\ell_c\), and the entry label \(\ell_{\text{in}}\)
  - \((\ell_{\text{out}}; \ell_r)\) the flow from the exit label \(\ell_{\text{out}}\) to the calling procedure \(\ell_r\).
Example

proc p(val x, res y) is\(^{\text{in}}\) S end\(^{\text{out}}\);

proc fib(val: z,u; res: v) is\(^{1}\)
    if \([z<3]\)\(^{2}\)
        then \([v:=u+1]\)\(^{3}\)
    else
        [call fib(z-1,u,v)]\(^{4}\) ; [call fib(z-2,v,v)]\(^{6}\)
        end\(^{8}\);
    [call fib(x,0,y)]\(^{9}\)\(^{10}\)
The flow graph

- [call fib(x,0,y)]
- [z<3]
- [v:=u+1]
- end

- is

- [call fib(z-2,v,v)]
- [call fib(z-1,u,v)]
The resulting flattened flow graph

\[ \text{[call fib}(x,0,y)\text{]}^{9}_{10} \]

\[ [z<3]^2 \]

\[ [v:=u+1]^3 \]

\[ \text{is}^1 \]

\[ \text{end}^8 \]

\[ [\text{call fib}(z-1,u,v)]^4_5 \]

\[ [\text{call fib}(z-2,v,v)]^6_7 \]
A naif approach

- We may simply extend the dataflow equations using the extended flow

\[
\begin{align*}
GA_{\odot}(\ell) &= \begin{cases} 
1 & \text{if } \ell \in E \\
\text{lub} \{ GA_{\odot}(\ell') \mid (\ell', \ell) \in F \text{ or } (\ell'; \ell) \in F \} & \text{otherwise}
\end{cases}
\end{align*}
\]

\[
GA_{\odot}(\ell) = f_{\ell} ( \text{GA}_{\odot}(\ell) )
\]
Correctness and Accuracy issues

• As we consider all possible paths \((l', l) \in F\) and \((l'; l) \in F\) the analysis is still correct.

• However, the analysis also consider the path \([9, 1, 2, 4, 1, 2, 3, 8, 10]\) that does not correspond to any actual computation of the program.

• This deeply affects the accuracy of the analysis.
The path [9, 1, 2, 4, 1, 2, 3, 8, 10] never occurs in the actual computations
Inter-flow

We may define a notion of inter-flow:

\[ \text{inter-flow} = \{(l_c, l_{in}, l_{out}, l_r) \mid \text{the program contains both } \]

\[ \text{[call } p(a,z)]^{l_c}_{l_r} \]

\[ \text{and } \text{proc } p(\text{val } x, \text{res } y) \text{ is }^{l_{in}} S \text{ end}^{l_{out}} \} \]
Flow and inter-flow

- flow = \{(1,2), (2,3), (2,4), (3,8), (4,1), (5,6), (6,1), (7,8), (8,5), (8,7), (8,10), (9,1)\}

- Inter-flow = \{(9,1,8,10), (4,1,8,5), (6,1,8,7)\}
Extending the general framework

\[ EA_\circ(\ell) = f_\ell \left( EA_\circ(\ell) \right) \]
for all labels \( \ell \) that do not appear as a first or last element of an inter-flow tuple

\[ EA_\circ(\ell) = \bigcup \{ EA_\circ(\ell') \mid (\ell', \ell) \in F \text{ or } (\ell'; \ell) \in F \} \cup \nu_\ell \]
for all labels \( \ell \)

Moreover, for each inter-flow tuple \((\ell_c, \ell_{in}, \ell_{out}, \ell_r)\) we introduce the equations:

\[ EA_\circ(\ell_c) = f_{\ell_c}( EA_\circ(\ell_c) ) \]

\[ EA_\circ(\ell_r) = f_{\ell_c, \ell_r}( EA_\circ(\ell_c), EA_\circ(\ell_r) ) \]