Flow Analysis

Data-flow analysis, Control-flow analysis, Abstract interpretation, AAM
Helpful Reading:
Sections 1.1-1.5, 2.1
Data-flow analysis (DFA)

- A framework for statically proving facts about program data.
  - Focuses on simple, finite facts about programs.
- Necessarily over- or under-approximate (*may* or *must*).
  - Conservatively considers all possible behaviors.
- Requires control-flow information; i.e., a control-flow graph.
  - If imprecise, DFA may consider infeasible code paths!
- Examples: reaching defs, available expressions, liveness,…
define fact(n : int) 
{
    s := 1;
    while (n > 1) 
    {
        s := s*n;
        n := n-1;
    }
    return s;
}
Control-flow graphs (CFGs)

- Intraprocedural CFGs may have a single entry/exit.

- Nodes can be a full **basic block** or a single statement.

- Each block or statement has $1^+$ predecessor and $1^+$ successor (stmt or block).

- A **fork point** is where paths diverge and a **join point** is where paths come together.
Data-flow analysis

• Computed by propagating facts *forward* or *backward*.

• Computes *may* or *must* information.

• Reaching definitions/assignments (def-use info): which assignments may reach each variable reference (use).

• Liveness: which variables are still needed at each point.

• Available expressions: which expressions are already stored.

• Very busy expressions: which expressions are computed down all possible paths forward.
Reaching defs, worklist algorithm

- Reaching definitions is a **forward may** analysis.
- gen(s) yields facts generated by a statement s.
- kill(s) yields facts invalidated by a statement s.
- pred(s) and succ(s) yield sets of statements preceding/succ.
- entry(s) = \( \bigcup_{s' \in \text{pred}(s)} \text{exit}(s') \); exit(s) = (entry(s) \ kill(s)) \cup\ gen(s)

- gen, kill, pred, succ, are fixed for each s; exit is monotonic in entry (if entry(s) grows, exit(s) grows); entry for exits of preds.
- We iterate rules for entry/exit until reaching a **fixed point**.
Reaching defs, worklist algorithm

• Main idea: worklist-based fixed-point algorithm.

• All entry(s) and exit(s) are initialized to be empty.

• Add all statements to the worklist; all must be considered.

• Until the worklist is empty, remove an s from worklist:
  • Compute entry(s) as union of all exit(s’), of predecessor s’
  • Compute exit(s) from gen(s), kill(s), and entry(s)
  • If exit(s) was increased, add all succ(s) to the worklist

• (This version is for forward may kill/gen analyses.)
Reaching defs, worklist algorithm

exit(s) = ∅
W = all statements s  //worklist
while W not empty:
    remove s from W

entry(s) = \bigcup_{s' \in \text{pred}(s)} \text{exit}(s')

update = (\text{entry}(s) \setminus \text{kill}(s)) \cup \text{gen}(s)
if update != \text{exit}(s):
    \text{exit}(s) = \text{update}
    \text{W} = \text{W} \cup \text{succ}(s)

Forward may analysis
## Reaching definitions analysis

<table>
<thead>
<tr>
<th>Stmt</th>
<th>GEN</th>
<th>KILL</th>
</tr>
</thead>
<tbody>
<tr>
<td>(s := 1;^1)</td>
<td>((s,1))</td>
<td>((s,*))</td>
</tr>
<tr>
<td>(n &gt; 1^2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(s := s*n;^3)</td>
<td>((s,3))</td>
<td>((s,1)), ((s,3))</td>
</tr>
<tr>
<td>(n := n-1;^4)</td>
<td>((n,4))</td>
<td>((n,0)), ((n,4))</td>
</tr>
<tr>
<td>return (s;^5)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Diagram](image-url)
\textbf{Reaching definitions analysis}

\begin{align*}
(n, 0) & \quad (s, 1) & (n, 4) & (s, 3) \\
(n, 0) & (s, 1) & (n, 4) & (s, 3) \\
(n, 0) & (s, 3) & (n, 4) \\
(n, 4) & (s, 3)
\end{align*}
Reaching definitions analysis

all(x) = {(x, ℓ) | ∀ℓ}

\[
\begin{align*}
RD_{\text{entry}}(1) &= RD_{\text{exit}}(\emptyset) \\
RD_{\text{entry}}(2) &= RD_{\text{exit}}(1) \cup RD_{\text{exit}}(4) \\
RD_{\text{entry}}(3) &= RD_{\text{exit}}(2) \\
RD_{\text{entry}}(4) &= RD_{\text{exit}}(3) \\
RD_{\text{entry}}(5) &= RD_{\text{exit}}(2)
\end{align*}
\]

\[
\begin{align*}
RD_{\text{exit}}(\emptyset) &= \{(n, \emptyset)\} \\
RD_{\text{exit}}(1) &= RD_{\text{entry}}(1) \setminus \text{all}(s) \\
&\quad \cup \{(s, 1)\} \\
RD_{\text{exit}}(2) &= RD_{\text{entry}}(2) \\
RD_{\text{exit}}(3) &= RD_{\text{entry}}(3) \setminus \text{all}(s) \\
&\quad \cup \{(s, 3)\} \\
RD_{\text{exit}}(4) &= RD_{\text{entry}}(4) \setminus \text{all}(n) \\
&\quad \cup \{(n, 4)\} \\
RD_{\text{exit}}(5) &= RD_{\text{entry}}(5)
\end{align*}
\]
Lattices

- Facts range over **lattices**: partial orders with **joins** (least upper bounds) and **meets** (greatest lower bounds).

```
{(s,1), (s,3), (n,4)}
{(s,1), (s,3)}  {(s,1), (n,4)}  {(s,3), (n,4)}
{(s,1)}  {(s,3)}  {(n,4)}
∅  ⊥
```
Lattices

• Facts range over lattices: partial orders with joins (least upper bounds) and meets (greatest lower bounds).

• A partial order is a set $X$ and an ordering $(X, \sqsubseteq)$ that is:
  • Reflexive; $\forall x. x \sqsubseteq x$
  • Transitive; $\forall x,y,z. x \sqsubseteq y \land y \sqsubseteq z \Rightarrow x \sqsubseteq z$
  • Anti-symmetric; $\forall x,y. x \sqsubseteq y \land y \sqsubseteq x \Rightarrow x = y$

• Lattices must also have unique joins and meets for any 2 points. Complete lattices have unique joins/meets for any set.

• Cartesian product of lattices is a lattice. Map with a lattice co-domain is a lattice.
Reaching definitions analysis

• The 11 sets $\text{RD}_{\text{exit}}(\emptyset)$, $\text{RD}_{\text{entry}}(1)$, $\text{RD}_{\text{exit}}(1)$,...$\text{RD}_{\text{exit}}(5)$, are defined in terms of one another. Written as a vector of sets:
  $$\vec{\text{RD}}$$

• Our set of equations can be turned into a **monotonic** $F$:
  $$\vec{\text{RD}}_0 \subseteq \vec{\text{RD}}_1 \Rightarrow F(\vec{\text{RD}}_0) \subseteq F(\vec{\text{RD}}_1)$$

• So that a satisfying vector of reachable defs is a **fixed point**:
  $$\vec{\text{RD}} = F(\vec{\text{RD}}) = F^n(\bot) \text{ for some } n$$

• For example, the join point would end up encoded as:
  $$F(..., \text{RD}_{\text{exit}}(1), ..., \text{RD}_{\text{exit}}(4), ...) = (... , \text{RD}_{\text{exit}}(1) \cup \text{RD}_{\text{exit}}(4), ...)$$
Very busy expressions analysis

- Computes a set of expressions that are computed down all paths forward before any subexpressions change value.

- Assignments represent GEN for the right hand side and KILL for expressions containing the right hand side (assigned var).

- Is a **backward must** data-flow analysis:
  - Propagates a set of computed expressions *backward*.
  - Computes the **meet** (GLB, intersection) of entry(s’) in $s’ \in \text{succ}(s)$ at each fork point to obtain exit(s).
Very busy expressions analysis

<table>
<thead>
<tr>
<th>Stmt</th>
<th>GEN</th>
<th>KILL</th>
</tr>
</thead>
<tbody>
<tr>
<td>s := 1;(^1)</td>
<td>1</td>
<td>s</td>
</tr>
<tr>
<td>n &gt; 1(^2)</td>
<td></td>
<td>s*n</td>
</tr>
<tr>
<td>s := s*n;(^3)</td>
<td>s*n</td>
<td>s</td>
</tr>
<tr>
<td>n := n-1;(^4)</td>
<td>n-1</td>
<td>n-1</td>
</tr>
<tr>
<td>return s;(^5)</td>
<td></td>
<td>s*n</td>
</tr>
</tbody>
</table>

**Diagram:**
- **fact(n)\(^0\) entry**
  - **s := 1;\(^1\)**
  - **n > 1\(^2\)**
    - **s := s*n;\(^3\)**
    - **n := n-1;\(^4\)**
  - **return s;\(^5\)**
- **fact(n) exit**
Very busy expressions analysis

\[
\text{fact}(n) \overset{0}{\text{entry}} \\
\quad s := 1;^1 \\
\quad n > 1^2 \\
\quad s := s \ast n;^3 \\
\quad n := n - 1;^4 \\
\quad \text{return } s;^5 \\
\text{fact}(n) \overset{\emptyset}{\text{exit}}
\]

1
∅
\[s \ast n, \ n - 1\]
∅
n - 1
∅
<table>
<thead>
<tr>
<th></th>
<th>May</th>
<th>Must</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward</td>
<td>Forward, computes exit(s) from entry(s)</td>
<td>Forward, computes exit(s) from entry(s)</td>
</tr>
<tr>
<td></td>
<td>Join (∪) at CFG join points</td>
<td>Meet (∩) at CFG join points</td>
</tr>
<tr>
<td></td>
<td>e.g., Reaching Defs (use-def) (which assignments reach uses)</td>
<td>e.g., Available Expressions</td>
</tr>
<tr>
<td>Backward</td>
<td>Backward, computes entry(s) from exit(s)</td>
<td>Backward, computes entry(s) from exit(s)</td>
</tr>
<tr>
<td></td>
<td>Join (∪) at CFG fork points</td>
<td>Meet (∩) at CFG fork points</td>
</tr>
<tr>
<td></td>
<td>e.g., Live Variables</td>
<td>e.g., Very Busy Expressions</td>
</tr>
</tbody>
</table>
exit(s) = ⊥
W = all statements s  //worklist
while W not empty:
    remove s from W

entry(s) = \bigcup_{s' \in \text{pred}(s)} \text{exit}(s')

update = (entry(s) \setminus \text{kill}(s)) \cup \text{gen}(s)
if update != exit(s):
    exit(s) = update
    W = W \cup \text{succ}(s)

Forward may analysis
exit(s) = \top // except \bot at function entry
W = all statements s //worklist
while W not empty:
    remove s from W

entry(s) = \bigcap_{s' \in \text{pred}(s)} \text{exit}(s')

update = (entry(s) \setminus \text{kill}(s)) \cup \text{gen}(s)
if update != \text{exit}(s):
    \text{exit}(s) = \text{update}
W = W \cup \text{succ}(s)

\textbf{Forward must analysis}
entry(s) = ⊥
W = all statements s //worklist
while W not empty:
    remove s from W

    exit(s) = \bigcup_{s' \in \text{succ}(s)} \text{entry}(s')

    update = (\text{exit}(s) \setminus \text{kill}(s)) \cup \text{gen}(s)
    if update \neq \text{entry}(s):
        \text{entry}(s) = \text{update}
        W = W \cup \text{succ}(s)

Backward may analysis
entry(s) = T // except ⊥ at function exit
W = all statements s // worklist
while W not empty:
    remove s from W
    exit(s) = \bigcap_{s' \in \text{succ}(s)} entry(s')
    update = (exit(s) \setminus \text{kill}(s)) \cup \text{gen}(s)
    if update != entry(s):
        entry(s) = update
        W = W \cup \text{succ}(s)

Backward must analysis
Abstract interpretation

• A general methodology for **justifying** or **calculating** sound analyses, given a precise semantics for the target language.

• Abstract interpretation establishes **abstract** semantic domains and a **Galois connection** between concrete and abstract.

• A function alpha ($\alpha: X \rightarrow \hat{X}$) defines a notion of abstraction.

• A function gamma ($\gamma: \hat{X} \rightarrow X$) defines a corresponding notion of concretization ($\alpha$ implies $\gamma$ and vice versa; more on this…).

• A concrete interpreter ($F: X \rightarrow X$) and Galois connection can be used to justify or calculate an abstract interpretation:

$$\alpha \circ F \circ \gamma \subseteq \hat{F}$$
Abstraction/Concretization (Galois)

\[ \alpha(x) \subseteq \hat{x} \]

if and only if

\[ x \subseteq \gamma(\hat{x}) \]
Abstraction/Concretization (Galois conn.)
Abstraction/Concretization (Galois conn.)

Values

\{\ldots,-1,0,1,\ldots\}
\{1,2,3,\ldots\}
\{1\} \quad \{2\} \quad \{3\} \ldots

Simple Types

Int

\alpha(2) = \text{Pos-Int}

\alpha

\gamma

\gamma

\gamma

\gamma
Constant Propagation

- Forward must style of DFA. Or as an abstract interpretation:

- Uses a flat lattice of constants with top and bottom (C, ⊑):

  $\top \rightarrow \cdots \rightarrow \top$
  $\perp \rightarrow \cdots \rightarrow \perp$

  $0 \rightarrow \cdots \rightarrow \top$
  $\top \rightarrow \cdots \rightarrow \perp$

  $1 \rightarrow \cdots \rightarrow \top$
  $\top \rightarrow \cdots \rightarrow \perp$

  $2 \rightarrow \cdots \rightarrow \top$
  $\top \rightarrow \cdots \rightarrow \perp$

  $\cdots$...

  “a” $\rightarrow \cdots \rightarrow \top$
  $\top \rightarrow \cdots \rightarrow \perp$

  #f $\rightarrow \cdots \rightarrow \top$
  $\top \rightarrow \cdots \rightarrow \perp$

  void $\rightarrow \cdots \rightarrow \top$
  $\top \rightarrow \cdots \rightarrow \perp$

- Facts become sets of pairs (Var x C) or a map (Env: Var → C).
Flow analysis

- Intraprocedural analysis: considers functions independently.
- Interprocedural analysis: considers multiple functions together.
- Whole-program analysis: considers an entire program at once.
- DFA is great for simple, local-variable-focused analyses.
- Analysis of heap-allocated data is much harder.
- The simple case is called pointer analysis (aliases, nullable).
- The general case is called shape analysis (full data-structures).
What about Scheme or ANF/CPS IRs?
What value can \( f \) be?

Data-flow depends upon control-flow.

Depends on call sites for the lambda that binds it.
Where does control propagate from this call-site?

\[(\text{lambda} \ (k \ f \ x \ y) \ (\text{let} \ ([a \ (\text{prim} + x \ y)]) \ (f \ k \ a)))\]

Depends on the possible values of parameter \(f\).

Control-flow depends upon data-flow.
The higher-order control-flow problem:
Data-flow and control-flow properties are thoroughly entangled and mutually dependent.
The solution?

*Control-flow analysis:* Simultaneously model control-flow behavior and data-flow behavior in a single analysis.
Control-flow analysis

• Use abstract interpretation to produce a single uniform analysis of all interdependent program properties.

• CFAs are whole-program interprocedural analyses. (Shivers 1991)

• $k$-CFA tracks $k$ latest call-sites; bindings have context. (EXPTIME)

• 0-CFA produces a single model of each function. ($0(\ell n^3))$

• Abstracting abstract machines (AAM): a unified methodology for deriving CFAs from concrete (precise) abstract machines! (Van Horn and Might 2010)
CPS lambda calculus

e \in \text{Exp} ::= (ae \ ae \ ...)

ae \in \text{AExp} ::= x \mid \text{lam}

lam \in \text{Lam} ::= (\text{lambda} \ (x \ ... \ e))

x \in \text{Var} \text{ is a set of variables}
CPS lambda calculus (semantics)

**DOMAINS**

State: Exp x Env

Env: Var → Clo

Clo: Lam x Env

**ATOMIC EVAL**

\[ A(lam, env) = (lam, env) \]

\[ A(x, env) = env(x) \]

**SMALL-STEP TRANSITION**

\[ ((ae_0 \ldots ae_j), env) \rightarrow (e_\theta, env'), \text{ where} \]

\[ \left( \text{((lambda } (x_1 \ldots x_j) e_\theta), env_c) \right) = A(ae_\theta) \]

\[ clo_i = A(ae_i) \]

\[ env' = env_c[x_i \mapsto clo_i] \]
Collecting semantics (generates a trace)
Env: Var $\rightarrow$ Clo

CPS store-passing semantics

**DOMAINS**

State: \(\text{Exp} \times \text{Env} \times \text{X} \times \text{Store}\)

Env: \(\text{Var} \rightarrow \text{Addr}\)

Store: \(\text{Addr} \rightarrow \text{Clo}\)

Clo: \(\text{Lam} \times \text{Env}\)

Addr: some infinite set

**ATOMIC EVAL**

\[ A(\text{lam}, \text{env}, \text{st}) = (\text{lam}, \text{env}) \]

\[ A(x, \text{env}, \text{st}) = \text{st}(\text{env}(x)) \]
CPS store-passing semantics

**SMALL-STEP TRANSITION**

$$((ae_0 \ldots ae_j), \text{env}, \text{st}) \rightarrow (e_0, \text{env}', \text{st}')$$,

where

$$((\lambda (x_1 \ldots x_j) e_0), \text{env}_c) = A(ae_0)$$

$$\text{clo}_i = A(ae_i)$$

$$\text{env}' = \text{env}_c[x_i \rightarrow \text{alloc}(x_i)]$$

$$\text{st}' = \text{st}[\text{alloc}(x_i) \rightarrow \text{clo}_i]$$

$$\text{alloc}(x_i) = \text{fresh address}$$

$$= x_i \text{ (yields } \emptyset\text{-CFA)}$$
Now we may finitize the set Addr

\[
\text{State: } \text{Exp} \times \text{Env} \times X \times \text{Store} \\
\text{Env: } \text{Var} \rightarrow \overset{\text{Addr}}{\text{Addr}} \\
\text{Store: } \overset{\text{Addr}}{\text{Addr}} \rightarrow \text{Clo} \\
\text{Clo: } \text{Lam} \times \text{Env}
\]
Now we may finitize the set $Addr$

$\hat{\text{State}}: \text{Exp} \times \hat{\text{Env}} \times X \times \hat{\text{Store}}$

$\hat{\text{Env}}: \text{Var} \rightarrow \hat{\text{Addr}}$

$\hat{\text{Store}}: \hat{\text{Addr}} \rightarrow \hat{\text{Clo}}$ (o)

$\hat{\text{Clo}}: \text{Lam} \times \hat{\text{Env}}$
Abstract abstract machines

\[
(f \ x, \ y \rightarrow a_y, \ f \rightarrow a_f, \ [x \rightarrow a_x, \ a_x \rightarrow \{\text{Int}\}, \ a_y \rightarrow \{\text{Int}\}, \ a_f \rightarrow \{\lambda (w) e_1, \ (\lambda (z) e_2)\}])
\]
Abstract-state transition

\[
\begin{align*}
(f \ x) &\quad [x \rightarrow a_x, \\
y &\rightarrow a_y, \\
f &\rightarrow a_f] \\
\end{align*}
\]

\[
\begin{align*}
[a_x &\rightarrow \{\text{Int}\}, \\
a_y &\rightarrow \{\text{Int}\}, \\
a_f &\rightarrow \{(\lambda (w) \ e_1), \\
&\quad (\lambda (z) \ e_2)\}] \\
\end{align*}
\]
Soundness

can be understood through the abstraction and transition between concrete and abstract representations.
Exponential complexity

(..., store)

addresses

values

✔ ✔ ✔
✔ ✔ ✔
✔ ✔
✔ ✔
Global store widening
(flow-sensitive)

Control-flow graph

per-point stores
Global store widening
(flow-insensitive)

Control-flow graph

Global store
Flow-insensitive (CPS) 0-CFA

Control-flow graph (of call-sites)

Variables map to sets of lambdas
Let’s live-code this 0-CFA