Flow Analysis

Data-flow analysis, Control-flow analysis, Abstract interpretation, AAM
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: reachingdefs, 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) \ \text{kill}(s)) \cup \text{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

\[
\begin{align*}
\text{exit}(s) &= \emptyset \\
W &= \text{all statements } s \quad \text{//worklist} \\
\text{while } W \text{ not empty:} \\
&\quad \text{remove } s \text{ from } W \\
\text{entry}(s) &= \bigcup_{s' \in \text{pred}(s)} \text{exit}(s') \\
\text{update} &= (\text{entry}(s) \setminus \text{kill}(s)) \cup \text{gen}(s) \\
\text{if } \text{update} \neq \text{exit}(s): \\
&\quad \text{exit}(s) = \text{update} \\
W &= W \cup \text{succ}(s)
\end{align*}
\]

Forward may analysis
# Reaching definitions analysis

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

```
fact(n)⁰ entry

s := 1;¹

n > 1²

s := s*n;³

n := n-1;⁴

return s;⁵

fact(n) exit
```
Reaching definitions analysis

\[(n, 0)\]

\[(n, 0) (s, 1) (n, 4) (s, 3)\]

\[(n, 0) (s, 1) (n, 4) (s, 3)\]

\[(n, 0) (s, 3) (n, 4)\]

\[(n, 4) (s, 3)\]
Reaching definitions analysis

all(x) = {(x, ℓ) | ∀ℓ}

\[ RD_{entry}(1) = RD_{exit}(\emptyset) \]
\[ RD_{entry}(2) = RD_{exit}(1) \cup RD_{exit}(4) \]
\[ RD_{entry}(3) = RD_{exit}(2) \]
\[ RD_{entry}(4) = RD_{exit}(3) \]
\[ RD_{entry}(5) = RD_{exit}(2) \]

\[ RD_{exit}(\emptyset) = \{(n, \emptyset)\} \]
\[ RD_{exit}(1) = RD_{entry}(1) \setminus all(s) \]
\[ \cup \{(s, 1)\} \]
\[ RD_{exit}(2) = RD_{entry}(2) \]
\[ RD_{exit}(3) = RD_{entry}(3) \setminus all(s) \]
\[ \cup \{(s, 3)\} \]
\[ RD_{exit}(4) = RD_{entry}(4) \setminus all(n) \]
\[ \cup \{(n, 4)\} \]
\[ RD_{exit}(5) = RD_{entry}(5) \]
Lattices

- Facts range over **lattices**: partial orders with **joins** (least upper bounds) and **meets** (greatest lower bounds).
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 $RD_{exit}(\emptyset), RD_{entry}(1), RD_{exit}(1), \ldots RD_{exit}(5)$, are defined in terms of one another. Written as a vector of sets:
  $$\vec{RD}$$

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

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

- For example, the join point would end up encoded as:
  $$F(\ldots, RD_{exit}(1), \ldots, RD_{exit}(4), \ldots) = (\ldots, RD_{exit}(1) \cup RD_{exit}(4), \ldots)$$
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’∈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;</td>
<td>1</td>
<td>s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>s*n</td>
</tr>
<tr>
<td>n &gt; 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s := s*n;</td>
<td>s*n</td>
<td>s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>s*n</td>
</tr>
<tr>
<td>n := n-1;</td>
<td>n-1</td>
<td>n-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>s*n</td>
</tr>
<tr>
<td>return s;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Very busy expressions analysis

\[ s := 1; \]
\[ n > 1 \]
\[ s := s \times n; \]
\[ n := n - 1; \]
\[ \text{return } s; \]

Diagram:

- **entry**
  - \[ s := 1; \]
  - \[ n > 1 \]
  - \[ s := s \times n; \]
  - \[ n := n - 1; \]
  - \[ \text{return } s; \]

- **exit**
## May/Must & Forward/Backward

<table>
<thead>
<tr>
<th></th>
<th>May</th>
<th>Must</th>
</tr>
</thead>
</table>
| **Forward** | Forward, computes exit(s) from entry(s)  
Join (∪) at CFG join points  
e.g., Reaching Defs (use-def)  
(which assignments reach uses) | Forward, computes exit(s) from entry(s)  
Meet (∩) at CFG join points  
e.g., Available Expressions |
| **Backward** | Backward, computes entry(s) from exit(s)  
Join (∪) at CFG fork points  
e.g., Live Variables | Backward, computes entry(s) from exit(s)  
Meet (∩) at CFG fork points  
e.g., Very Busy Expressions |
exit(s) = \bot
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 != \text{exit}(s):
    \text{exit}(s) = \text{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)
    \text{if update} \neq \text{exit}(s):
        \text{exit}(s) = \text{update}
        W = W \cup \text{succ}(s)

\text{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)} entry(s')

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

Backward may analysis
entry(s) = ⊤ // 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)} \text{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 ($\alpha: X \to \hat{X}$) defines a notion of abstraction.

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

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

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

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

if and only if

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

\[ Y(\hat{x}) \xrightarrow{\alpha} \alpha(x) \]

\[ X \xrightarrow{\square} \hat{X} \xrightarrow{Y} Y(x) \]
Abstraction/Concretization (Galois conn.)

Values

{..., -1, 0, 1, ...} ⊑ {1, 2, 3, ...} ⊑ {1} ⊑ {2} ⊑ {3} ...

Simple Types

Int

∪

Pos-Int

∪

α(2) = Pos-Int

α
Constant Propagation

- Forward must style of DFA. Or as an abstract interpretation.
- Uses a flat lattice of constants with top and bottom (C, ⊑):

  \[
  \begin{array}{ccccccc}
  \top & \rightarrow & \rightarrow & \rightarrow & \rightarrow & \rightarrow & \rightarrow \\
  \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
  0 & 1 & 2 & \ldots & "a" & \#f & \text{void} \\
  \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
  \bot & \rightarrow & \rightarrow & \rightarrow & \rightarrow & \rightarrow & \rightarrow \\
  \end{array}
  \]

- 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?
(\lambda (k \ f \ x \ y)
  (let ([a (prim + x y)])
    (f k a)))

What value can f be?

Data-flow depends upon control-flow.

Depends on call sites for the lambda that binds it.
Depends on the possible values of parameter $f$.

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

Where does control propagate from this call-site?

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. ($0(n^3)$)

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

\[ e \in \text{Exp} ::= (ae \ ae \ \ldots) \]
\[ ae \in \text{AExp} ::= x \mid \text{lam} \]
\[ \text{lam} \in \text{Lam} ::= (\text{lambda} \ (x \ \ldots) \ e) \]
\[ x \in \text{Var} \text{ is a set of variables} \]
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**

\[
((ae_0 \ldots ae_j), \text{env}) \rightarrow (e_0, \text{env}')
\]

where

\[
((\text{lambda } (x_1 \ldots x_j) e_0), \text{env}_c) = A(ae_\theta)
\]

\[
clo_i = A(ae_i)
\]

\[
\text{env}' = \text{env}_c[x_i \rightarrow clo_i]
\]
Collecting semantics (generates a trace)
Env: Var → Clo

Exp x Env

CPS store-passing semantics

**DOMAINS**

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

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

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

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

Addr: some infinite set

**ATOMIC EVAL**

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

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

**SMALL-STEP TRANSITION**

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

where \[ ((\lambda (x_1 \ldots x_j) e_0), \text{env}_c) = A(ae_0) \]
\[ \text{clo}_i = A(ae_i) \]
\[ \text{env'} = \text{env}_c[x_i \rightarrow \text{alloc}(x_i)] \]
\[ \text{st'} = \text{st}[\text{alloc}(x_i) \rightarrow \text{clo}_i] \]

\[ \text{alloc}(x_i) = \text{fresh address} = x_i \text{ (yields 0-CFA)} \]
Now we may finitize the set Addr

\[
\text{State: Exp} \times \text{Env} \times \text{X} \times \text{Store}
\]

\[
\text{Env: Var} \rightarrow \overline{\text{Addr}}
\]

\[
\text{Store: \overline{\text{Addr}} \rightarrow \text{Clo}}
\]

\[
\text{Clo: Lam} \times \text{Env}
\]
Now we may finitize the set $\text{Addr}$

\[
\text{State: } \text{Exp} \times \text{Env} \times \text{X} \times \text{Store} \\
\text{Env: } \text{Var} \rightarrow \text{Addr} \\
\text{Store: } \text{Addr} \rightarrow \wp(\text{Clo}) \\
\text{Clo: } \text{Lam} \times \text{Env}
\]
Int ⊑ Zero

abstraction
Abstract abstract machines

\[
\begin{align*}
(f & \ x) \\
& \quad , \quad \begin{cases}
 x \rightarrow a_x, \\
y \rightarrow a_y, \\
f \rightarrow a_f
\end{cases}, \\
\quad \begin{cases}
 a_x \rightarrow \{\text{Int}\}, \\
a_y \rightarrow \{\text{Int}\}, \\
a_f \rightarrow \{(\lambda \ (w) \ e_1), \\
(\lambda \ (z) \ e_2)\}\}
\end{cases}
\end{align*}
\]
Abstract-state transition

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

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

concrete transition

abstraction

abstract transition

abstraction
Exponential complexity

(..., store)

addresses

{ values

✔ ✔ ✔

✔ ✔ ✔

✔
Global store widening
(flow-insensitive)

Control-flow graph

Global store