DATA FLOW ANALYSIS  

Conservative approximation to global information on data flow properties 
that are relevant for optimizations 

→ MAY-problems vs. MUST-problems 

Examples: 

- Constant Propagation Analysis 
  Has \textit{var} always the same constant value at this point? 

- Reaching Definitions 
  Which definitions of \textit{var} may be relevant for this use? 

  - local (BB) 
  - global (CFG) using “effects” of entire BB’s (summary info) 
    forward vs. backward, iterative vs. interval-based vs. structured... 
  - interprocedural 

Example: Reaching Definitions (cont.) 

Definition \( d \) of variable \( v \): \( d; v \leftarrow ... \) 

\( d \) reaches a point \( p \) in CFG 
if there is a path \( d \rightarrow^* p \) in CFG (excl. \( d, p \)) 
that contains no kill of \( d \) (= reassignment of \( v \)) 

NB: Whether a specific definition \( d \) actually reaches a specific program point \( p \) is undecidable in the formal 
sense! 
(program behavior may e.g. depend on run-time input) 

→ conservative approximation 
  MAY-REACH or MUST-REACH, depending on the application. 

Background: Bitvector representation of sets 

Given: Finite global set (universe) \( U \) 
Any subset \( S \subseteq U \) can be represented as a bitvector \( b_S \) 
with \( b_S[i] = 1 \) iff the \( i \)th element of \( U \) is in \( S \). 

Example: 
\( U = \{a, b, c, d, e, f, g, h\} \) 
\( S = \{a, d, e\} \) has bitvector representation \( b_S = \langle 10011000 \rangle \). 

If clear from the context, we simplify the notation, using \( S \) for \( b_S \): 
\( S = \langle 10011000 \rangle \). 

Here: Consider bitvector representation of \textit{sets of definitions} 
i.e., the universe \( U \) = the set of all definitions in the program 
= set of all CFG nodes (e.g. MIR statements) writing to some variable.
Example (cont.): Bitvector Representation of Definitions; \textit{GEN} sets

\begin{itemize}
  \item \textbf{Bit Definition (generated) Basic block}
  \begin{enumerate}
    \item $d_1$ of \textit{m} in node 1 \hspace{1cm} B1
    \item $d_2$ of $f_0$ in node 2
    \item $d_3$ of $f_1$ in node 3
    \item $d_4$ of $i$ in node 4 \hspace{1cm} B3
    \item $d_5$ of $f_2$ in node 8 \hspace{1cm} B6
    \item $d_6$ of $f_0$ in node 9
    \item $d_7$ of $f_1$ in node 10
    \item $d_8$ of $i$ in node 11
  \end{enumerate}

  \begin{align*}
  GEN(B1) &= \{d_1,d_2,d_3\} = \langle 11100000 \rangle \\
  GEN(B3) &= \{d_4\} = \langle 00010000 \rangle \\
  GEN(B6) &= \{d_5,d_6,d_7,d_8\} = \langle 00011111 \rangle \\
  GEN(Bi) &= \{\} = \langle 00000000 \rangle \\
  \text{for } i \neq B1, B3, B6
  \end{align*}

\end{itemize}

Example: Reaching definitions with bitvector representation

\begin{enumerate}
  \item $RDin(B) = \langle 00100010 \rangle$ \hspace{1cm} (1 = def. reaches entry of B)

  Certainly, $RDin[entry] = \langle 00000000 \rangle$

  $RDin(B)$ for $B \neq entry$ ?

  Effect of a node $B$ in CFG on definitions $d$ reaching it:

  described by 2 \textit{sets} $GEN(B)$, $KILL(B)$:

  $GEN(B) = \langle 11100000 \rangle$ \hspace{1cm} (1 if $B$ generates this definition)

  $KILL(B) = \langle 11100010 \rangle$ \hspace{1cm} (1 if $B$ kills this definition)

  $RDout(B) = \langle 11100000 \rangle$ \hspace{1cm} (1 = def. reaches end of $B$, $?=bit as in RDin(B)$

  Example: $RDin(B) = \langle 10001101 \rangle$ and effect of $B$ as above

  $\implies RDout(B) = \langle 11101001 \rangle$

\end{enumerate}

Example (cont.): Bitvector Representation of Definitions; \textit{KILL} sets

\begin{itemize}
  \item \textbf{Bit Definition (generated) Basic block}
  \begin{enumerate}
    \item $d_1$ of \textit{m} in node 1 \hspace{1cm} B1
    \item $d_2$ of $f_0$ in node 2
    \item $d_3$ of $f_1$ in node 3
    \item $d_4$ of $i$ in node 4 \hspace{1cm} B3
    \item $d_5$ of $f_2$ in node 8 \hspace{1cm} B6
    \item $d_6$ of $f_0$ in node 9
    \item $d_7$ of $f_1$ in node 10
    \item $d_8$ of $i$ in node 11
  \end{enumerate}

  \begin{align*}
  KILL(B1) &= \{d_1,d_2,d_3,d_4\} = \langle 11100110 \rangle \\
  KILL(B3) &= \{d_4,d_5\} = \langle 00010001 \rangle \\
  KILL(B6) &= \{d_2,d_3,d_4,d_5,d_6,d_7,d_8\} = \langle 01111111 \rangle \\
  KILL(Bi) &= \{\} = \langle 00000000 \rangle \\
  \text{for } i \neq B1, B3, B6
  \end{align*}

\end{itemize}

Flow functions — Effect of a basic block $B$ on any $RDin(B)$:

\begin{itemize}
  \item \textbf{Set equation:}$RDout(B) = GEN(B) \cup (RDin(B) \setminus KILL(B)) \hspace{1cm} \forall B$
  \item \textbf{Bitvector equation:}$RDout(B) = GEN(B) \lor (RDin(B) \land \neg KILL(B)) \hspace{1cm} \forall B$
\end{itemize}

Effect of joining control flow paths:

\begin{itemize}
  \item \textbf{Set equation:}$RDin(B) = \bigcup_{P \in \text{Pred}(B)} RDin(P) \hspace{1cm} \forall B$ (for MUST-REACH: $\subseteq$)
  \item \textbf{Bitvector equation:}$RDin(B) = \bigvee_{P \in \text{Pred}(B)} RDin(P) \hspace{1cm} \forall B$ (for MUST-REACH: $\land$)
\end{itemize}

Reaching Definitions is a forward flow problem:

- BB flow functions specify outgoing property as function of ingoing
- Information propagates through CFG in direction from \textit{entry} towards \textit{exit}
Iterative computation of Reaching Definitions

Algorithm: (Fixed-point iteration)

- For MAY-Reach we initialize
  \[ RDin(\text{entry}) = \{\} = (00000000), \]
  \[ RDin(B) = \{\} = (00000000) \]
  for all other \( B \)

- Iterate,
  applying the equations
to \( RDin(B) \), \( RDout(B) \) for all \( B \)
until no more changes occur.

Example: see whiteboard

Why does this work?

---

Example (cont.): Iterative computation of Reaching Definitions

**Second iteration:**

\[
\begin{align*}
RDin(\text{entry}) &= (00000000) \\
RDout(\text{entry}) &= (00000000) \\
RDin(B1) &= (00000000) \\
RDout(B1) &= (11100000) \\
RDin(B2) &= (11100000) \\
RDout(B2) &= (11100000) \\
RDin(B3) &= (11100000) \\
RDout(B3) &= (11110000) \\
RDin(B4) &= (11111000) \\
RDout(B4) &= (11110000) \\
RDin(B5) &= (11111100) \\
RDout(B5) &= (11110000) \\
RDin(B6) &= (11110000) \\
RDout(B6) &= (10011111) \\
RDin(\text{exit}) &= (11111000) \\
RDout(\text{exit}) &= (11111000)
\end{align*}
\]

---

Example (cont.): Iterative computation of Reaching Definitions

**Third iteration:**

\[
\begin{align*}
RDin(\text{entry}) &= (00000000) \\
RDout(\text{entry}) &= (00000000) \\
RDin(B1) &= (00000000) \\
RDout(B1) &= (11100000) \\
RDin(B2) &= (11100000) \\
RDout(B2) &= (11100000) \\
RDin(B3) &= (11100000) \\
RDout(B3) &= (11110000) \\
RDin(B4) &= (11111000) \\
RDout(B4) &= (11110000) \\
RDin(B5) &= (11111100) \\
RDout(B5) &= (11110000) \\
RDin(B6) &= (11110000) \\
RDout(B6) &= (10001111) \\
RDin(\text{exit}) &= (11111100) \\
RDout(\text{exit}) &= (11111100)
\end{align*}
\]

No more change — done!
Why does this work?

Underlying theory:
- Posets, least upper bounds, semilattices, lattices
- Monotone flow functions
- Data flow analysis framework
- Meet-over-all-paths
- Convergence theorems for iterative data flow analysis

Posets

A relation \( \subseteq \) on a set \( L \) defines a partial order on \( L \) if, for all \( x, y \) and \( z \) in \( L \),
1. \( x \subseteq x \) (reflexive),
2. If \( x \subseteq y \) and \( y \subseteq x \) then \( x = y \) (antisymmetric), and
3. If \( x \subseteq y \) and \( y \subseteq z \) then \( x \subseteq z \) (transitive).

The pair \((L, \subseteq)\) is called a poset or partially ordered set.

Notation: \( x \subseteq y \) iff \( x \subsetneq y \).

Example: \( L = 2^S \) for a set \( S \), \( \subseteq = \supseteq \)
Interpretation in data flow analysis: \( x \subseteq y \) means "\( x \) is not more precise than \( y \)"

Least upper bound, greatest lower bound

A greatest lower bound (glb) of any two elements \( x, y \in L \) is an element \( g \in L \) such that
1. \( g \subseteq x \),
2. \( g \subseteq y \), and
3. for any \( z \in L \) with \( z \subseteq x \) and \( z \subseteq y \), \( z \subseteq g \).

Example: For \((2^S, \supseteq)\), glb is set union \((\cup)\).

Analogously: Least upper bound (lub).

A poset \((L, \subseteq)\) where any two elements in \( L \) have a greatest lower bound in \( L \) (i.e., closedness under glb) is a necessary condition for a semilattice.

Semilattice

A semilattice \((L, \sqcap)\) consists of a set \( L \) and a binary meet operator \( \sqcap \) such that for all \( x, y \in L \),
1. \( x \sqcap x = x \) (meet is idempotent),
2. \( x \sqcap y = y \sqcap x \) (meet is commutative),
3. \( x \sqcap (y \sqcap z) = (x \sqcap y) \sqcap z \) (meet is associative),

and there is a top element \( \top \in L \) such that
4. for all \( x \in L \), \( \top \sqcap x = x \).

Optionally, a semilattice may also have a bottom element \( \bot \in L \) with for all \( x \in L \), \( \bot \sqcup x = \bot \).

Example 1: \((2^S, \cup)\) is a semilattice with \( \top = \{\} \) and \( \bot = S \).
Example 2: \((2^S, \cap)\) is a semilattice with \( \top = S \) and \( \bot = \{\} \).
Semilattice and partial order

A semilattice \((L, \sqcap)\) implicitly defines a partial order \(\sqsubseteq\) where, for all \(x, y \in L\),
\[ x \sqsubseteq y \iff x \sqcap y = x. \]
The glb is just the \(\sqcap\) operator.

Example 1: \((2^x, \cup)\) implicitly defines partial order \(\subseteq\).
Example 2: \((2^x, \cap)\) implicitly defines partial order \(\subseteq\).

Note: \(\bot \not\subseteq x \not\subseteq \top\) for all \(x \not\subseteq \top\), \(x \not\subseteq \bot\).

Interpretation: \(\top\) is most precise information, \(\bot\) is most imprecise information.

Example: Bitvector Lattice

Bitvector lattice: \(L = BV^3\), \(\sqcap = \) union/bitwise OR, \(\sqcup = \) interse./bitwise AND

Partial order \(\sqsubseteq\):
\[ x \sqsubseteq y \iff x \sqcap y = x \]
(transitive, antisymmetric, reflexive)
for all \(x\): \(\bot \not\subseteq x \subseteq \top\)

meet \(x \sqcap y\): follow paths in \(L\) from \(x, y\) downwards until they meet
(greatest lower bound w.r.t. \(\sqsubseteq\))

join \(x \sqcup y\): follow paths in \(L\) from \(x, y\) upwards until they join
(least upper bound w.r.t. \(\sqsubseteq\))

Lattice \((L, \sqcap, \sqcup)\)

- set \(L\) of values
- meet operation \(\sqcap\), join operation \(\sqcup\) where
  - (1) for all \(x, y \in L\) ex. unique \(z, w \in L\): \(x \sqcap y = z, x \sqcup y = w\) (closedness)
  - (2) for all \(x, y \in L\): \(x \sqcap y = y \sqcap x, x \sqcup y = y \sqcup x\) (commutativity)
  - (3) for all \(x, y, z \in L\): \((x \sqcap y) \sqcap z = x \sqcap (y \sqcap z), (x \sqcup y) \sqcup z = x \sqcup (y \sqcup z)\) (associativity)
  - (4) there are two unique elements of \(L\):
    \[ \top \text{ “top”: } \forall x \in L, x \sqcup \top = \top \]
    \[ \bot \text{ “bottom”: } \forall x \in L, x \sqcap \bot = \bot \]
  - (5) often also distributivity of \(\sqcap, \sqcup\) given

Lattices: Monotonicity, Height; Termination

\(f : L \rightarrow L\)

is monotone iff \(\forall x, y \in L\): \(x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)\)

Example:
\(f : BV^3 \rightarrow BV^3\) with \(f((x_1x_2x_3)) = (x_1.1x_3)\) for all \(x_1, x_2, x_3 \in BV\) is monotone.
\(g : BV^1 \rightarrow BV^1\) with \(g(0) = 1\) and \(g(1) = 0\) is not monotone.

Height of \((L, \sqcap, \sqcup)\)
\[ = \text{ length of longest strictly ascending chain in } L \]
\[ = \max. n: \exists x_1, x_2, \ldots, x_n \in L \text{ with } \bot = x_1 \sqsubseteq x_2 \sqsubseteq \ldots \sqsubseteq x_n = \top \]

Example:
Height of \(BV^3\) is 4.

Finite height + Monotonicity \(\Rightarrow\) Termination of the fixed-point iteration
**Flow functions**

Flow functions specify the **effect** of a programming language construct as a mapping $L \rightarrow L$.

E.g., in Reaching Definitions:

- $BB_B_1$ generates $d_1, d_2, d_3$, kills $d_1, d_2, d_3, d_6, d_7$:
  
  $F_{B_1}(\langle x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 \rangle) = \langle 111 x_4 x_5 00 x_8 \rangle$

- $F_{B_1}(\langle x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 \rangle) = \langle x_1 x_2 x_3 1 x_4 x_5 0 \rangle$

- $F_{B_6}(\langle x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 \rangle) = \langle x_1 0001111 \rangle$

- $F_{B_j} = id$ for all $j \notin \{1, 3, 6\}$

Flow functions must be monotone.

(otherwise the fixed-point iteration algorithm could oscillate)

---

**Fixed points**

Fixed point of a function $f : L \rightarrow L$ is a $z \in L$ with $f(z) = z$

- Solution to a set of data flow equations

  - In general not unique!

  **Example:**

  $f : BV \rightarrow BV$ with $f(0) = 0$ and $f(1) = 1$

  has 2 fixed points: 0 and 1.

Reaching definitions (see above):

iterate until $f(RDin(B)) = RDin(B) \forall B$

where $f = \text{composition of all flow functions and equations}$.

---

**The ideal solution**

**Ideal solution (IDEAL)** to the data flow equations (for forward problems):

- begin with initial information $Init$ at **entry**

- apply composition of flow functions along all **really** possible paths from **entry** to each CFG node $B$

  and compose these results by the meet operator:

$IDEAL(B) = \bigcap_{p \in \text{Paths}(B)} F_p(Init)$

similarly for backward problems

---

**Meet over all paths (MOP)**

**Meet-over-all-paths (MOP) solution** to data flow equations (for forward problems):

- begin with initial information $Init$ at **entry**

- apply composition of flow functions along **all** possible paths from **entry** to each CFG node $B$

  and compose these results by the meet operator:

$MOP(B) = \bigcap_{p \in \text{Paths}(B)} F_p(Init)$

similarly for backward problems
MOP vs. IDEAL

A solution $in$ is **safe** if $in(B) \subseteq IDEAL[B] \forall B$
A solution $in$ is **incorrect** if $in(B) \not\subseteq IDEAL[B]$ for some $B$

BUT: **IDEAL** is statically undecidable!

The exact subset of the paths really taken at run time is not statically known. E.g., an else branch or loop may never be executed.

**IDEAL**($B$) = Meet over all paths to $B$ possibly taken at run time
NEVER($B$) = Meet over all remaining paths to $B$ (never executed)

The most precise solution is **IDEAL**($B$), but **MOP**($B$) = **IDEAL**($B$) \text{\inter} NEVER($B$), i.e., **MOP**($B$) \not\subseteq IDEAL($B$).

**MOP** is the best solution that we could compute statically.

Fixed point solutions of the dataflow equations

Goal: find the maximum fixed point (MFP) solution
(maximal w.r.t. information, i.e., also w.r.t. $\subseteq$, and still safe)

**Theorem** [Kildall’73]

If all flow functions distributive over $\sqcap$, $\sqcup$

i.e., $\forall x,y$, $f(x \sqcap y) = f(x) \sqcap f(y)$ and $f(x \sqcup y) = f(x) \sqcup f(y)$,

$\Rightarrow$ iterative DFA computes MFP, and MFP = **MOP**

**Theorem** [Kam/Ullman’75]

If all flow functions monotone but not necessarily distributive

$\Rightarrow$ iterative DFA computes MFP but not necessarily the MOP solution

### Iterative Data Flow Analysis [Kildall’73]

given: CFG $G = (N,E)$, Lattice $(L,\sqcap,\sqcup)$

dataflow equations

\[
in(B) = \begin{cases} 
\text{Init} & \text{for } B = \text{entry} \\
\bigcap_{P \in \text{Pred}(B)} out(P) & \text{otherwise} 
\end{cases} 
\]

\[
out(B) = F_B(in(B)) 
\]

or, by substitution,

\[
in(B) = \begin{cases} 
\text{Init} & \text{for } B = \text{entry} \\
\bigcap_{P \in \text{Pred}(B)} F_P(in(P)) & \text{otherwise} 
\end{cases} 
\]

\text{Init} is usually $\top$ (for $\sqcap$) or $\bot$ (for $\sqcup$)

### Iterative DFA: Worklist algorithm (1)

- Implements the fixed-point algorithm above
- Maintain a **worklist** of blocks $B$ whose predecessors’ $in$ values have changed in the last iteration
- worklist contains initially all BB’s (except entry)
- iterate applying the dataflow equations until no more changes occur

Observation: maximal effect on forwarding information
if BB’s in worklist are processed in topological order

$\Rightarrow$ start with reverse postorder
$\Rightarrow$ queue as worklist
$\Rightarrow A + 2$ iterations for a (sub-)CFG with $A$ back edges [Hecht/Ullman’75]
Iterative DFA: Worklist algorithm (2)

```
Worklist_it \( N, \text{entry} \), \( F \), \( DFin \), \( Init \)
Set\(<\text{Node}> N; \)
Node \( \text{entry} \); \nFunctions \( F : \text{Node} \times L \rightarrow L \); \nFunction \( DFin : \text{Node} \rightarrow L \); \n\( \text{L Init} : \text{L} \) is the (semi-)lattice \( \{ L \text{ totaleffect}, \text{effectP} \}; \)
List\(<\text{Node}> W \leftarrow N \setminus \{ \text{entry} \}; \)
\( DFin[\text{entry}] \leftarrow \text{Init}; \)
for each \( B \in N \) do
  \( DFin(B) \leftarrow \top; \)
  \( \) 
 repeat \n  \( B \leftarrow W.\text{delete_first_element}(); \)
  \( \text{totaleffect} \leftarrow \top; \)
  for each \( P \in \text{Pred}(B) \) do
    \( \text{effectP} \leftarrow F(P, DFin(P)); \)
    \( \text{totaleffect} \leftarrow \text{totaleffect} \sqcap \text{effectP}; \)
    if \( DFin(B) \neq \text{totaleffect} \) then
      \( DFin(B) \leftarrow \text{totaleffect}; \)
      \( W \leftarrow W \cup \text{Succ}(B); \)
  until \( W = \emptyset; \)
return \( DFin; \)
```

Survey of some data flow problems

classified by:
- information to be computed
- direction of information flow: forward / backward / bidirectional
- lattices used, meanings attached to lattice elements etc.

Reaching Definitions
forward, bitvector (1 bit per definition of a variable)

Available Expressions
forward, bitvector (1 bit per definition of an expression)

Live Variables
backward, bitvector (1 bit per use of a variable)

Survey of some data flow problems (cont.)

Upwards Exposed Uses
backward, bitvector (1 bit per use of a variable)

Copy-Propagation Analysis
forward, bitvector (1 bit per copy assignment)

Constant-Propagation Analysis
forward, \( ICP^m \) (or similar)
1 lattice value per def., symbolic execution

Partial Redundancy Analysis
[Morel,Renvoise’81] bidirectional, bitvector (1 bit per expression computation)
[Knoop/Rüthing/Steffen’92] “Lazy Code Motion”

Available Expressions

An expression, say \( x+y \), is \textit{available} at a point \( p \) if:
(1) \textit{every} path from the \textit{entry} node to \( p \) evaluates \( x+y \), and
(2) after the last evaluation prior to reaching \( p \),
there are no subsequent assignments to \( x \) or \( y \).

We say that a basic block \textit{kills} expression \( x+y \)
if it \textit{may} assign \( x \) or \( y \), and does not subsequently recompute \( x+y \).
Live Variables

A variable is **live** at a program point \( p \) if there is a path from \( p \) to any use of \( v \) that does not contain a definition of \( v \).

**Flow problem:** backward, bitvector (1 bit per use of a variable)

Upwards Exposed Uses

A use \( u \) of a variable \( v \) is **upwards exposed** at a program point \( p \) if there is a path from \( p \) to \( u \) that does not contain a definition of \( v \).

**Flow problem:** backward, bitvector (1 bit per use of a variable)

Copy Propagation Analysis

A copy statement \( x \leftarrow y \) assigns variable \( y \) to \( v \).

Can we safely replace all occurrences of \( x \) by \( y \), in order to eliminate the copy statement and variable \( x \) completely?

**Flow problem:** forward, bitvector (1 bit per copy assignment)

Constant Propagation Analysis

**Flow problem:** forward analysis, using \( ICP^m \) (or similar)

(1 lattice value per definition, symbolic execution)

\[ ICP: \]

\[ x \leftarrow \text{const} \]

\[ \text{false} \leftarrow -2 \leftarrow -1 \leftarrow 0 \leftarrow 1 \leftarrow 2 \leftarrow \text{true} \]

(undefined)

(not a constant)
Partial Redundancy Elimination

bidirectional, bitvector: 1 bit per expression computation

[Morel, Renvoise'81] bidirectional, bitvector (1 bit per expression computation)
[Knoop/Rüthing/Steffen'92] “Lazy Code Motion”
[Dhamdhere'02] “PRE made easy”

Web Construction Example

5 webs (sets of intersecting DU-chains):

\{
\{ (x, (B2, 1)) \},
\{ (x, (B3, 1)), (B5, 1) \}
\}
\{ (y, (B4, 1)) \}
\{ (z, (B5, 1)) \}
\{ (x, (B6, 2)), (B6, 1) \}
\{ (z, (B6, 1)) \}

sparse representation of dataflow information about variables:

- DU-chain connects a definition to all uses it may reach
- UD-chain connects a use to all definitions that may reach it

implemented as lists

Web for a variable \( v \)

= maximal union of intersecting DU-chains for \( v \)

useful in global register allocation (count as one live range)

DU, UD chains are implicitly given in SSA form (→).

Structural Dataflow Analysis — Example: Reaching Definitions

\( GEN[R] = \{ d \} \)
\( KILL[R] = \{ d ; d, \text{ defines } a \} \)
\( RDout[R] = GEN[R] \cup (RDim[R] \setminus KILL[R]) \)
\( GEN[R] = GEN[R2] \cup (GEN[R1] \setminus KILL[R2]) \)
\( KILL[R] = KILL[R2] \cup (KILL[S1] \setminus GEN[R2]) \)
\( RDim[R1] = RDim[R] \)
\( RDim[R2] = RDout[R2] \)
\( RDout[R] = RDout[R2] \)
\( GEN[R] = GEN[R1] \cup GEN[R2] \)
\( KILL[R] = KILL[R1] \cup KILL[R2] \)
\( RDim[R1] = RDim[R] \)
\( RDim[R2] = RDout[R2] \)
\( RDout[R] = RDout[R1] \cup RDout[R2] \)
\( GEN[R] = GEN[R1] \)
\( KILL[R] = KILL[R1] \)
\( RDim[R1] = RDim[R] \cup GEN[R1] \)
\( RDout[R] = RDout[R1] \)
Data Flow Analysis: Summary

- Gather global information about data flow properties
- Safe under-/overestimation, depending on intended transformations
- Propagation over the CFG → iterative data flow analysis, implemented with the Worklist algorithm
- Lattice theory:
  - Monotonicity + Finite height ⇒ Termination of fixed-point iteration
- Various data flow problems and methods
- DU/UD chains, webs
- Structural dataflow analysis

Data Flow Analysis, further topics and outlook:

- Further DFA methods (interval / structural analysis)
- Array data flow analysis [Feautrier’91], [Maydan/Hennessy/Lam’91]
- DFA for pointers and heap data structures
- SSA form
- Generators for Data Flow Analyzers, e.g. Sharlit [Tjiang/Hennessy’92], PAG [Martin’98]