Abstract Interpretation

Welf Löwe
Welf.Lowe@lnu.se

Complete Partial Order (CPO)

- Partially ordered sets $(\mathcal{C}, \subseteq)$ over a universe $U$
- Smallest element $\bot \in U$
- Partial order relation $\subseteq$
- Ascending chain $C = \{c_0, c_1, \ldots \} \subseteq U$
  - Smallest element $c_0$
  - $c_n \subseteq c_{n+1}$
  - Maybe finite or countable; constructor for next element $c_{n+1} = \text{next}(c_0, c_1, \ldots, c_n)$
- Unique largest element $\upsilon$ of chain $C$:
  - $\nu = \nu$ (larger than all chain elements $c_i$)
  - called supremum $\nu = \text{sup}(C)$

- Ascending chain property: any (may be countable) ascending chain $C \subseteq U$ has an element $c_0$ with
  - $\nu$ is finite and
  - for all elements $c_i \subseteq c_0$
  - for all elements $c_i = c_0$
  - then $\nu = \text{sup}(C)$

Example: $\{0,1\}$ and $C = \{0,1\}$, $\{1,2\}$, $\{1,2,3\}$… $C^\omega = \text{max}(C_i) \uparrow \cup C_{i-1}$ $\nu = \text{sup}(C) = \mathbb{N}$ but ascending chain property does not hold

CPOs and Lattices

- Lattice $\mathcal{L} = (\mathcal{U} \uplus \mathcal{F})$
  - any two elements $a, b$ of $\mathcal{U}$ have
    - an infimum $\mathcal{g}(a, b)$ - unique largest smaller of $a, b$
    - a supremum $\mathcal{u}(a, b)$ - unique smallest bigger of $a, b$
  - unique smallest element $\bot$ (bottom)
  - unique largest element $\top$ (top)
- A lattice $\mathcal{L} = (\mathcal{U} \uplus \mathcal{F})$ defines two kinds of CPOs $(\mathcal{U}, \subseteq)$
  - upwards:
    - $a \subseteq b \Rightarrow \mathcal{u}(a, b) = b$ smallest $\bot$
    - finite heights $\Rightarrow$ ascending chain property holds ($\subseteq = \bot$)
  - downwards:
    - $b \subseteq a \Rightarrow \mathcal{g}(a, b) = a$ smallest $\bot$
    - finite heights $\Rightarrow$ ascending chain property holds ($\subseteq = \bot$)

Functions on CPOs

- Functions $f : U \rightarrow U'$ (if not indicated otherwise, we assume $U = U'$)
- $f$ monotone: $a \subseteq b \Rightarrow f(a) \subseteq f(b)$ with $a, b \in U$
- $f$ continuous: $\mathcal{u}(\mathcal{C}) = \mathcal{u}(f(C))$ with $\mathcal{C} = \{f(v_0), \ldots, f(v_n)\}$
- $f$ continuous $\Rightarrow f$ monotone,
- $f$ monotone $\Rightarrow f$ continuous
- $f$ monotone $\Rightarrow \mathcal{U}, \subseteq$ a CPO with ascending chain property $\Rightarrow f$ continuous
- $f$ monotone $\Rightarrow \mathcal{U}, \subseteq$ a lattice with finite heights $\Rightarrow f$ continuous
Example

- \( U = \mathcal{P}(N) \) (set of all subsets of Natural numbers \( N \))
- Define:
  - \( f_a(x) = \emptyset \cup u \in U \) finite
  - \( f_a(x) = N \cup u \in U \) infinite
  - \( f \) is monotone: \( \subseteq \rightarrow \subseteq \)
  - \( \emptyset \subseteq \emptyset \subseteq \cdots \)
  - \( f \) is not continuous: \( f(U(x)) \neq f(x) \), e.g.:
- \( C = \{[0], \{0,1\}, \ldots \} \)
  - \( C^* = \emptyset \cup \{0\} \cup \{0,1\} \cup \{0,1,2\} \cup \ldots \)
  - \( f(C) = \{0\} \cup f(\{0\}) \cup \{0,1\} \cup \ldots \cup f(\{0,1,2,\ldots\}) \)
- \( f(U(C)) = \emptyset \cup f(C) \cup \{0\} \cup \{0,1\} \cup \ldots = \emptyset \cup \emptyset \cup \emptyset \cup \ldots \)
- \( f(U(C)) = f(U(\{0\})) \cup f(\{0,1\}) \cup \ldots = \emptyset \cup \emptyset \cup \ldots \)
- \( f(\emptyset) = \emptyset \)
- \( f(\{0\}) = \{0\} \)
- \( f(\{0,1\}) = \{0,1\} \)
- \( f(\{0,1,2\}) = \{0,1,2\} \)
- \( f(\{0,1,2,\ldots\}) = \{0,1,2,\ldots\} = \emptyset \)
- \( f(N) = N \)
- \( \emptyset \not\subseteq f(N) \)
- \( \cup \subseteq \emptyset \cup f(\{0\}) \cup \{0,1\} \cup \cdots = \emptyset \cup \emptyset \cup \ldots \)

Fixed Point Theorem (Knaster-Tarski)

- Fixed point of a function: \( X = f(X) \)
- For \( CPO(U) \) and monotone functions \( f: U \rightarrow U \)
  - Minimum (or least or smallest) fixed point \( X \) exists
    - \( X \) is unique
- For \( CPO(U) \) with smallest element \( \bot \) and continuous functions \( f: U \rightarrow U \)
  - Minimum fixed point \( X = \bigcup f^n(U) \)

Monotone DFA Framework

- Solution of a set of DFA equations is a fix point computation
- Contribution of a computation \( \delta \) of kind \( k \) (Alloc, Add, Load, Store, Call, ...) is modeled by monotone transfer function
- \( \delta^k: U \rightarrow U \)
- Set of transfer functions is closed under composition and obviously the composed functions are monotone as well
- Contribution of predecessors \( Pre \) of \( \delta \) is modeled by supremum \( \cup \) of predecessor properties (successor Succ, resp., for backward problems)
- Existence of the smallest fix point \( X \) is guaranteed, if domain \( U \) of analysis values \( P(X) \) completely partially ordered (\( U \subseteq U \))
- It is efficiently computable if \( U \subseteq U \) additionally fulfills the ascending chain property

Monotone DFA Framework (cont’d)

- Data flow equations define monotone functions in \( (N \times U \times U \subseteq) \):
  - \( P_{\text{in}}(A) = \bigcup P_{\text{in}}(P_{\text{out}}(X)) \)
  - \( P_{\text{out}}(A) = f_{\text{out}}(P_{\text{out}}(X)) \) with \( f_{\text{out}}(\subseteq) \) transfer function of \( \delta \)
- Smallest fix point of this system of equations is efficiently computable since
  - \( (N \times U \times U \subseteq)^{\omega} \) completely partially ordered and fulfills the ascending chain property
- System of equations defines monotone function in \( (N \times U \times U \subseteq)^{\omega} \)
- Data flow analysis algorithm:
  - Start with the smallest element: \([([0], \bot, \bot), ([0], \bot, \bot), \ldots ([0], \bot, \bot)]\]
  - Apply equations in any (fair) order
  - Until no \( P_{\text{in}}(A) \) or \( P_{\text{out}}(A) \) changes

Monotone DFA Framework (cont’d)

- Forward and must:
  - \( P_{\text{in}}(A) \neq \emptyset \)
  - \( P_{\text{out}}(A) = \emptyset \)
- Backward and must:
  - \( P_{\text{out}}(A) = \emptyset \)
  - \( P_{\text{in}}(A) \neq \emptyset \)
- Forward and may:
  - \( P_{\text{out}}(A) \neq \emptyset \)
  - \( P_{\text{in}}(A) \neq \emptyset \)
- Backward and may:
  - \( P_{\text{out}}(A) \neq \emptyset \)
  - \( P_{\text{in}}(A) \neq \emptyset \)

4 DFA Equations Schemata

- \( P_{\text{in}}(A) \neq \emptyset \)
- \( P_{\text{out}}(A) = \emptyset \)
- \( P_{\text{out}}(A) \neq \emptyset \)
- \( P_{\text{in}}(A) \neq \emptyset \)
Initialization

- Assume a Power Set Lattice \(\mathcal{P}^X\)
- Initialization with the smallest element
- General initializations with \(\perp\) for all but start node \(n'\):
  - \(\mathbf{may}\): Initialization with \([\{n', e, \perp\}, \{n', \perp, \perp\}, \ldots, \{n', \perp\} \}]\) as empty set \(\perp\) is the smallest element for each position
  - \(\mathbf{must}\): Initialization with \([\{n', e, \perp\}, \{n', \perp, \perp\}, \ldots, \{n', \perp\} \}]\) as universe of values \(\perp\) is the smallest element for each position in the inverse lattice
- Special (problem specific) initializations \(\iota\):
  - \(\mathbf{forward}\): \([\ldots, \{n', e, \perp\}, \ldots]\) as general initialization not defined before the start node
  - \(\mathbf{backward}\): \([\ldots, \{n', e, \perp\}, \ldots]\) as general initialization not defined after the end node

Property \(P, x = 1\) possible?
- Universe Boolean, CPO Boolean Lattice
- Transfer functions identical
- Forward – may problem
  - \(\mathcal{P} \supseteq \mathcal{P}_N \supseteq \mathcal{P}_F\)
  - Begin with \(\mathcal{P}_{\infty} = \mathcal{P}\) (assumption \(x = 1\))
  - Initialization \(\mathcal{P}_{\infty} = \mathcal{P}\)
  - Iteration leads to fixed point \(\mathcal{P}_N = \mathcal{P}\)
- Generalization:
  - Compute properties of several (all) variables in each step
  - Property: are variables equal to a specific constant or are variables actually compile time constants at a certain program point
  - Universe: Bit vector with a vector element for each variable
  - CPO induced by bit vector lattice

What does Data Flow Analysis?
- Property: \(P, x = 1\) guaranteed?
- Universe Boolean, CPO Boolean Lattice
- Transfer functions: true, false, id
- Statement \(x := x + 1\)
- Statement \(y := y + 1\)
- Statement \(C, y := 1\) \(\iff\) does not change
- Let \(P, P_F, P_N, P_{\infty}\) be values of \(P\)
- After statement \(x := 1, C, y := 1\)
- Let \(P_{\infty} = P_N = P_F = P\)
- Initialization \(\mathcal{P}_\infty = \mathcal{P}\)
- Forward – must problem
  - \(x \leftarrow \mathbf{false}\)
  - Begin with \(\mathcal{P}_{\infty} = \mathcal{P}\) before statements (assumption \(x = 1\))
  - Initialization \(\mathcal{P}_{\infty} = \mathcal{P}\) before statement \(x = 1\)
  - Iteration leads to fixed point \(P_N = \mathcal{P}\)
- \(x = 1\) is more difficult:
  - Obviously, a naive transfer function \(x := 1\) is not monotone
  - Conservative transfer function \(x := 1\)
  - Conservatively, \(x = 1\) is not guaranteed any more by analysis in some cases where we (as humans) could see it holds

Path Graph
- For nodes \(n \in N\) of \(G(N, E)\) define path graph \(G'(n) = (N, E')\)
- For every path \(P\) ending in \(n\):
  - \(n' \in \Pi\) \(\Rightarrow\) \(n' \in N\)
  - \(\{n', n''\} \in E'\) \(\iff\) \(\{n', n''\} \in \Pi\)
- The path graph is acyclic by definition
- Since the set of paths to a node \(n\) in \(G\) is possibly countable (iff \(G\) contains loops) the graph \(G'(n)\) is in general not finite

Example I

Example II

Example: Path Graph
MFP and MOP

For a monotone DFA problem (set of equations) \( DFE = \langle U, F, \alpha \rangle \) and \( G \):
- Define: Minimum Fixed Point \( MFP \) is computed by iteratively applying \( F \) beginning with the smallest element in \( U \).
- Let \( DFE' = \langle U, F, \alpha \rangle \) and \( G' = \langle G, \alpha \rangle \) (same equations as \( DFE \), applied to path graphs).
- Define: Meet Over all Paths \( MOP \) of \( DFE' \) in (any arbitrary) node \( a \) is the minimum fix point, \( MFP \), of \( DFE' \) in node \( a \).
- \( MFP \) is equivalent with \( MOP \), if are distributive over \( \cup \) in \( U \).
- \( MFP \) is a conservative approximation of the \( MOP \) otherwise.

Attention:
- It is not decidable if a path is actually executable.
- Hence, \( MOP \) is already conservative approximation of the actual analysis result since some path may be not executable in any program run.
- \( MOP > MOP \) (meet over all executable paths)

Example for \( MFP(G) \neq MOP(G) \)

Constant propagation: value vector: \( (x,y,z) \in \{0,1,unknown\}^3 \)

\[ G \]
\[
\begin{array}{ccc}
(a,u,u) & \rightarrow & (u,u,0) \\
(0,1,0) & \rightarrow & (u,u,0) \\
(u,u,0) & \rightarrow & (a,u,u) \\
0 & \rightarrow & 0 \\
\end{array}
\]

\[ G' \]
\[
\begin{array}{ccc}
(a,u,u) & \rightarrow & (u,u,0) \\
(1,0,0) & \rightarrow & (u,u,0) \\
(u,u,0) & \rightarrow & (a,u,u) \\
0 & \rightarrow & 0 \\
\end{array}
\]

Errors due to our DFA Method

- Call Graphs:
  - Nodes = Procedures, Edges = calls
  - Only a conservative approximation of actually possible calls, some calls represented in the call graph might never occur in any program run.
  - Allows impossible paths like call → procedure → another call

- Data Flow Graphs:
  - Nodes = Statements (Expressions), Edges = (initial or essential) dependencies between them
  - Application of a monotone DFA framework computes \( MFP \), not \( MOP \)

Problems left open

- How to derive the transfer functions for a DFA
- How to make sure they compute the intended result, i.e.,
  - \( MOP \leq MFP \)?

Example: Reaching Definitions (Must)

- Which „Definitions“ (assignments) are guaranteed to reach a node \( x \)?
- Solution: Subset of all nodes \( \{A_1, \ldots, A_n\} \) containing an assignment to a variable reaching a access from this variable in node \( x \)

- Universe: e.g. a bit-vector representation \( \{false, true\} \ldots \{false, true\} \) of all possible subsets of nodes for each variable.
- Forward, Must
  - Schema: \( RD_f(A) = 3 \bigcup RD_f(A) \)
    \( RD_f(A) = RD_f(A) \cap kill(A) \cup gen(A) \)

  - Initialization:
    - Universe, i.e. all definitions \( \{(A_1, \ldots, A_n)\} \) reach each program point
    - Start node: no definition reaches, i.e. \( RD_f(A) = 0 \) for all variables

- A definition is generated \( (gen(A)) \) by assignment \( x:=expr \)
- A definition is removed \( (kill(A)) \) by an assignment \( x:=expr \) to the same variable in another node

Outline

- Summary of Data Flow Analysis (yesterday’s lecture)
- Problems left open
- Abstract interpretation idea

Example: Reaching Definitions (Must)

- Which „Definitions“ (assignments) are guaranteed to reach a node \( x \)?
- Solution: Subset of all nodes \( \{A_1, \ldots, A_n\} \) containing an assignment to a variable reaching a access from this variable in node \( x \)

- Universe: e.g. a bit-vector representation \( \{false, true\} \ldots \{false, true\} \) of all possible subsets of nodes for each variable.
- Forward, Must
  - Schema: \( RD_f(A) = 3 \bigcup RD_f(A) \)
    \( RD_f(A) = RD_f(A) \cap kill(A) \cup gen(A) \)

  - Initialization:
    - Universe, i.e. all definitions \( \{(A_1, \ldots, A_n)\} \) reach each program point
    - Start node: no definition reaches, i.e. \( RD_f(A) = 0 \) for all variables

- A definition is generated \( (gen(A)) \) by assignment \( x:=expr \)
- A definition is removed \( (kill(A)) \) by an assignment \( x:=expr \) to the same variable in another node
Idea even generalizes to other than dataflow analyses, as
allows to compute or prove correct data flow equations
How does this generalize?

How to make sure well (e.g., control flow analysis)
abstract analysis semantics

finite many contexts for each program point)
(infinitely many) possible dataflow problems
Prove that they are abstractions indeed.
Example run:

A: x
B: y

• A
• B

Problem left open

• How to make sure \( RD \) computes the correct result?
  • As intended by the problem
  • Exact result or a conservative approximation

Actually, in the example program and the specific run \( RD \) behaves correctly:
• Static analysis: \( RD_{\lambda}(N) = \emptyset \)
• Example run: \( RD_{\lambda}(N) = \{ A \} \)
  • Recall that \( RD \) was a meet problem, ascending on the downwards CPO induced by the lattice power set lattice
  • Hence \( \subseteq \) relation is the inverse set inclusion \( \supseteq \) on the label sets

How does this generalize?
• For all runs, all programs, and for all dataflow problems
  • We cannot test all (countable) paths of all (countable) programs and all (infinitely many) possible dataflow problems

Abstract Interpretation

• Relates semantics of a programming language to an abstract analysis semantics
• Allows to compute or prove correct data flow equations (transfer functions)
  • Define abstraction of the execution semantics wrt. analysis problem
  • Define abstraction of execution traces to program points (in general, finite many contexts for each program point)
  • Prove that they are abstractions indeed.
• Idea even generalizes to other than dataflow analyses, as well (e.g., control flow analysis)

Program Traces

Each program run is defined by a trace \( \iota \in Labels^* \)
Traces are defined by the programming language semantics, e.g.,
• \( n[\text{if expr then stats else stats}] = n[\text{if expr}] \cap n[\text{stats}] \cap n[\text{else stats}] \)
• \( n[\text{while expr do stats}] = n[\text{while}] \cap n[\text{expr}] \cap n[\text{do stats}] \)
For instance, actual reaching definitions \( RD_{\lambda} \), is a mapping \( RD_{\lambda} : Tr \rightarrow 2^{Labels} \)
i.e., for each trace \( \iota \) a subset of definitions \( \subseteq 2^{Labels} \) reaches the end point of that trace
Analysis Execution Semantics

- Non-standard semantics: actually expected analysis results are defined for traces as an abstraction of the program’s execution semantics wrt. the analysis problem
- Standard semantics: a program’s execution semantics is defined by the semantics of each programming language construct and their composition in the program
- There are only finitely many such constructs

Traces and semantics analysis values define a CPO \((U, \subseteq)\)
- Universe \(U\) defined by \(U \rightarrow 2^{\mathcal{Tr}}\)
- Partial order \(\subseteq\): same process, same semantics, same traces
- Smallest element \(\epsilon \rightarrow \emptyset\)
- Universe \(U\) is countable not finite
- Even if the semantic analysis function \((e.g., \ R_D)\) is monotone it is in general not continuous as universe not finite since \(\mathcal{Tr}(G)\) is not
- Then a solution to the analysis problem may exist, but cannot be computed iteratively by applying the analysis function on the smallest element to fix point
  - Non-terminating program runs
  - Infinitely many different inputs

RD_{apr} Execution Semantics

- Given a program \(G = (N, E, n^i)\)
- \(RD_{apr}: Tr \rightarrow 2^{\mathcal{Tr}}\)
- Basis for recursive definitions: empty trace
  - No definition reaches the end of the empty trace
- Analysis execution semantics of \(n \oplus label\) (trace \(n\) expanded by the next dynamic step \(label\))
  - recursively defined on analysis execution semantics of trace \(n\) and analysis execution semantics of the static programming language construct of step \(label\)
- \(RD_{apr}(\emptyset) = (\emptyset \oplus label)\)
- \(RD_{apr}(\emptyset) = (\emptyset \oplus label)\) if \((1, x := expr) \in N\) \(U (label)\)
  - else \(RD_{apr}(\emptyset)\)

Solution

- Define an abstraction \(\alpha\) of traces to make universe infinite
- Perform an abstract analysis on the abstraction of traces
- Define a inverse concretization function \(\gamma\) to map results back to the semantic domain of the programming language
- \(\alpha\) and \(\gamma\) should form a so called adjunction, or Galois connection, i.e.
  \(\alpha(\chi) \leq Y \Rightarrow X \subseteq \gamma(Y)\)
  - Mind the different domains of \(\alpha\) and \(\gamma\)
  - Consequently, there are different partial order relations \(\subseteq\)
  - \(\subseteq\) on analysis execution semantics domain, countable and
  - \(\subseteq\) on abstract analysis domain, finite
- Showing that abstraction and concretization form a Galois connection is one of our proof obligations to prove the analysis correct

Observation

- Traces and semantics analysis values define a CPO \((U, \subseteq)\)
  - Universe \(U\) defined by \(U \rightarrow 2^{\mathcal{Tr}}\)
  - Partial order \(\subseteq\): same program, same labels, and same traces
  - Smallest element \(\epsilon \rightarrow \emptyset\)
  - Universe \(U\) is countable not finite
- Even if the semantic analysis function \((e.g., \ R_D)\) is monotone it is in general not continuous as universe not finite since \(\mathcal{Tr}(G)\) is not
- Then a solution to the analysis problem may exist, but cannot be computed iteratively by applying the analysis function on the smallest element to fix point
  - Non-terminating program runs
  - Infinitely many different inputs

Galois Connections

\[ \alpha(X) \leq Y \Rightarrow X \subseteq \gamma(Y) \]

Countable Execution CPO \((U, \subseteq)\)
- Finite Abstraction CPO \((U', \leq)\)

How to define the analysis?

- Take (any) abstract analysis \(F\) function abstracting (i.e.,
gives larger results that) \(\alpha \circ Act \circ \gamma: U' \rightarrow U'\) where \(Act\) is the actual analysis execution semantics function
- Analysis terminates if \((U', \subseteq)\) a CPO and \(F\) monotone
- Analysis is conservative if \(Act\) is monotone and \((\alpha, \gamma)\) a Galois connection
- Then conservative approximation computable by fixed point iteration
- It holds for the minimum fix points \(MFP:\)
  \(\alpha(MFP(Act)) \leq MFP(\alpha \circ Act \circ \gamma) \leq MFP(F)\)
Reaching Definitions ($\alpha$)

- Given a program $G = (N, E, v')$
- $RD : Lables \rightarrow glabels$
- Basis for recursive definitions:
  - Empty trace abstraction: starting point of the program $v'$
  - no definition reaches $v'$
  - $RD_n(e') = \emptyset$
- Analysis semantics at label which is actually $\alpha \cdot RD_n \cdot \gamma$
- recursively defined on analysis semantics of abstractions of predecessor traces $n$ (predecessor labels)
- $RD_{label}(S) = \bigcup RD_{\alpha}(RD_{\gamma}(S))$
  - analysis abstraction of the execution semantics of the static programming language construct of step label (transfer function)
- $RD_{\alpha}(S) = \emptyset$
- $RD_{\gamma}(S) = \emptyset$
- Use structural induction over all programs
- Correctness of Analysis Abstraction
- Compare execution semantics and analysis semantics (transfer functions) of program constructs
- Basis:
  - Claim holds for the empty trace: each program's starting point is abstracted correctly: $RD_\alpha(e') = \emptyset, RD_\gamma(g) = \emptyset$
  - Step:
    - Given a trace $\tau \oplus label$ and its abstraction label
    - Provided $RD_\gamma(label)S$ is a correct abstraction of $RD_\gamma(\tau \oplus label)$
    - Then $RD_\gamma(label)S$ is a correct abstraction of $RD_\gamma(\tau \oplus label)$.
    - $\forall \tau \in label : RD_\alpha(\gamma(RD_\gamma(label))) \subseteq RD_\gamma(label)$
    - Distinguish cases of each program construct and transfer function
      - Here trivial as $RD_\alpha$ and $RD_\gamma$ are identical (and monotone)

General Proof Obligations

- To show (i): $(\alpha, \gamma)$ is a Galois connection (obvious)
- To show (ii): $\alpha \cdot RD_n \cdot \gamma$ is abstracted with RD i.e., $\alpha \cdot RD_n \cdot \gamma \subseteq RD$
- Proof (sketch): for each node $n$ of $G$
  - By our definition of $\gamma, \gamma(label)^G = TR_{label}$ of corresponds to path graph of $G$ in $n = (label)S$
  - By our definition of $RD_n, \alpha \cdot RD_n \cdot \gamma$ in a node $n$ is $MFP$
  - MFP of RD of path graph of $G$ in $n$ is $MOP$ of $G$ in $n$
  - $MOP \subseteq MFP$ of $RD$

RD Analysis Semantics

- Let $TR_{label}$ be the set of all traces ending with program point label
  - $TR_{label}(\tau \oplus label \in Label \times \tau \oplus label$ is an admissible trace of $G)$
- We abstract a set $TR_{full} = P^G$ with that program point label $label$
- Let $\tau \in P^G \rightarrow label$
- $TR_{label}(\tau) = label$
- Concrete and abstract analysis value domains $glabels$ are the same:
  - Let $RD_\alpha(\tau \oplus label) \in glabels$ be the set of definitions reaching label
  - Let $RD_\gamma(label) \in glabels$ be the set of reaching definitions analyzed for label
- Let $\alpha$ label $\gamma$
- We abstract the analysis execution semantics of the trace $\tau \in TR_{label}$ $\gamma$ with the abstract analysis results $RD_{label}(\tau)$ of the program point label
  - $\forall \tau \in TR_{label}$ $\alpha RD_{\alpha}(\tau) \rightarrow RD_{\gamma}(label)$
  - $\forall \tau \in TR_{label}$ $\alpha RD_{\alpha}(\tau) \rightarrow RD_{\gamma}(label)$

Reaching Definitions ($\gamma$)

- Conversely, we concretize each program point label with the set of all traces ending in label
- The concretization function on labels is
  - $\gamma : Label \rightarrow glabels$
- $\gamma(label) = TR_{label}$
- Consequently, we concretize the abstract analysis results $RD_{\gamma}(label)$ of a program point label by assuming it holds for any of the traces $\tau \in TR_{label}$
  - $\forall \tau \in TR_{label}$ $\gamma RD_{\gamma}(label) \rightarrow RD_{\gamma}(label)$