DATA DEPENDENCIES AND PROGRAM SLICING:

FROM SYNTAX TO ABSTRACT SEMANTICS

Isabella Mastroeni and Damiano Zanardini

PEPM 2008

Slicing vs dependencies

Slicing: ...extracts from programs the statements which are *relevant* for a given behaviour.

SLICING VS DEPENDENCIES

SLICING: ...extracts from programs the statements which are *relevant* for a given behaviour.

DEPENDENCY: ...defines what *relevant* means.

SLICING VS DEPENDENCIES

SLICING: ...extracts from programs the statements which are *relevant* for a given behaviour.

DEPENDENCY: ...defines what *relevant* means.

$$\text{SYNTACTIC DEF-REF}: \begin{cases} x := y + 2z \\ \mathbf{x} \text{ depends on } \mathbf{y} \text{ and on } \mathbf{z} \end{cases}$$

SLICING VS DEPENDENCIES

SLICING: ...extracts from programs the statements which are *relevant* for a given behaviour.

DEPENDENCY: ...defines what *relevant* means.

SYNTACTIC DEF-REF :
$$\begin{cases} x := y + 2z \\ \textbf{x} \text{ depends on } \textbf{y} \text{ and on } \textbf{z} \\ \\ x := z + y - y \\ \textbf{x} \text{ depends on } \textbf{y} \text{ and on } \textbf{z} \end{cases}$$

SLICING VS DEPENDENCIES

SLICING: ...extracts from programs the statements which are *relevant* for a given behaviour.

DEPENDENCY: ...defines what *relevant* means.

SEMANTIC : $\begin{cases} x := z + y - y \\ \textbf{x} \text{ depends on } \textbf{z} \text{ but it does NOT depend on } \textbf{y} \end{cases}$

SLICING VS DEPENDENCIES

SLICING: ...extracts from programs the statements which are *relevant* for a given behaviour.

DEPENDENCY: ...defines what *relevant* means.

$$\text{SEMANTIC}: \begin{cases} x := z + y - y \\ \textbf{x} \text{ depends on } \textbf{z} \text{ but it does NOT depend on } \textbf{y} \\ \\ x := 2y \\ \textbf{x} \text{ depends on } \textbf{y} \end{cases}$$

SLICING VS DEPENDENCIES

SLICING: ...extracts from programs the statements which are *relevant* for a given behaviour.

DEPENDENCY: ...defines what *relevant* means.

ABSTRACT SEMANTIC (PARITY) : $\begin{cases} x := 2y \\ \mathbf{x} \text{ does NOT depend on } \mathbf{y} \end{cases}$

SLICING VS DEPENDENCIES

SLICING: ...extracts from programs the statements which are *relevant* for a given behaviour.

DEPENDENCY: ...defines what *relevant* means.

ABSTRACT SEMANTIC (PARITY) : $\left\{ \begin{array}{l} x := 2y \\ \textbf{x} \text{ does NOT depend on } \textbf{y} \\ \\ x := 2y + z \\ \textbf{x} \text{ depends on } \textbf{z} \end{array} \right.$

RELATED WORKS

- Slicing by means of a calculus for independencies [Amtoft & Banerjee '07];
  - Syntactic dependencies
  - Forward slicing

RELATED WORKS

- Slicing by means of a calculus for independencies [Amtoft & Banerjee '07];
  - Syntactic dependencies
  - Forward slicing
- Abstract dependencies [Rival '05];
  - Mathematical, set theoretic definition of dependencies;
  - Applied to Alarm diagnosis;

Related works

- Slicing by means of a calculus for independencies [Amtoft & Banerjee ’07];
  - Syntactic dependencies
  - Forward slicing
- Abstract dependencies [Rival ’05];
  - Mathematical, set theoretic definition of dependencies;
  - Applied to Alarm diagnosis;
- Abstract Slicing [Hong et al. ’05]
  - Only for predicate abstractions;
  - Considers a subset of possible executions

Abstract Interpretation

Consider the complete lattice $< C, \leq, \wedge, \vee, \bot, \top >$, $A_i \in \mathsf{uco}(C)$

Lattice of Abstract Domains $\equiv$ Lattice uco

$\mathsf{A} \equiv \rho(C)$

$< \mathsf{uco}(C), \sqsubseteq, \sqcap, \sqcup, \lambda x.\, \top, \lambda x.\, x >$

ABSTRACT INTERPRETATION

Consider the complete lattice $< C, \leq, \wedge, \vee, \perp, \top >$, $A_i \in \mathrm{UCO}(C)$

Lattice of Abstract Domains $\equiv$ Lattice UCO

$\mathsf{A} \equiv \rho(C)$

$< \mathrm{UCO}(C), \sqsubseteq, \sqcap, \sqcup, \lambda x.\, \top, \lambda x.\, x >$

$$A_1 \sqsubseteq A_2 \quad \Leftrightarrow \quad A_2 \subseteq A_1$$

ABSTRACT INTERPRETATION

Consider the complete lattice $< C, \leq, \wedge, \vee, \perp, \top >$, $A_i \in \text{uco}(C)$

Lattice of Abstract Domains $\equiv$ Lattice uco

A $\equiv \rho(C)$

$< \text{uco}(C), \sqsubseteq, \sqcap, \sqcup, \lambda x.\, \top, \lambda x.\, x >$

$$A_1 \sqsubseteq A_2 \quad \Leftrightarrow \quad A_2 \subseteq A_1$$

$$\sqcap_i A_i = \mathcal{M}(\cup_i A_i)$$

ABSTRACT INTERPRETATION

Consider the complete lattice $< C, \leq, \wedge, \vee, \perp, \top >$, $A_i \in \mathrm{UCO}(C)$

Lattice of Abstract Domains $\equiv$ Lattice UCO

A $\equiv \rho(C)$

$< \mathrm{UCO}(C), \sqsubseteq, \sqcap, \sqcup, \lambda x.\, \top, \lambda x.\, x >$

$$A_1 \sqsubseteq A_2 \quad \Leftrightarrow \quad A_2 \subseteq A_1$$

$$\sqcap_i A_i = \mathcal{M}(\cup_i A_i)$$

$$\sqcup_i A_i = \cap_i A_i$$

Abstract Interpretation

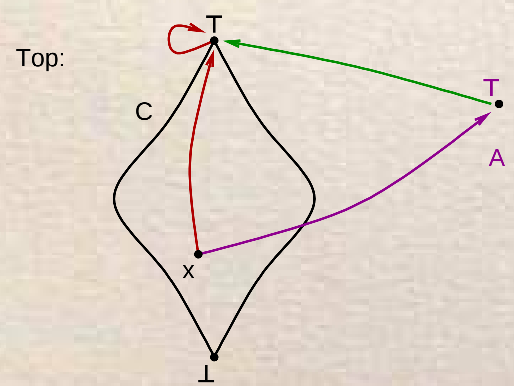
Consider the complete lattice $< C, \leq, \wedge, \vee, \perp, \top >$, $A_i \in \text{uco}(C)$

Lattice of Abstract Domains $\equiv$ Lattice uco

A $\equiv \rho(C)$

$< \text{uco}(C), \sqsubseteq, \sqcap, \sqcup, \lambda x.\, \top, \lambda x.\, x >$

$$A_1 \sqsubseteq A_2 \quad \Leftrightarrow \quad A_2 \subseteq A_1$$
$$\sqcap_i A_i = \mathcal{M}(\cup_i A_i)$$
$$\sqcup_i A_i = \cap_i A_i$$



Abstract Interpretation

Consider the complete lattice $< C, \leq, \wedge, \vee, \perp, \top >$, $A_i \in \mathrm{uco}(C)$

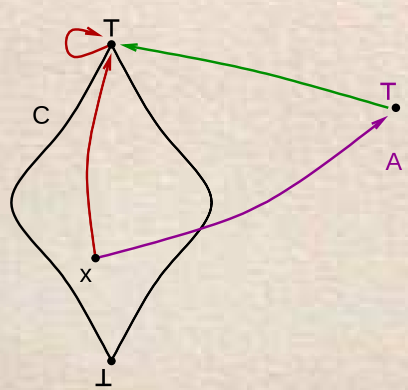
Lattice of Abstract Domains $\equiv$ Lattice uco

A $\equiv \rho(C)$

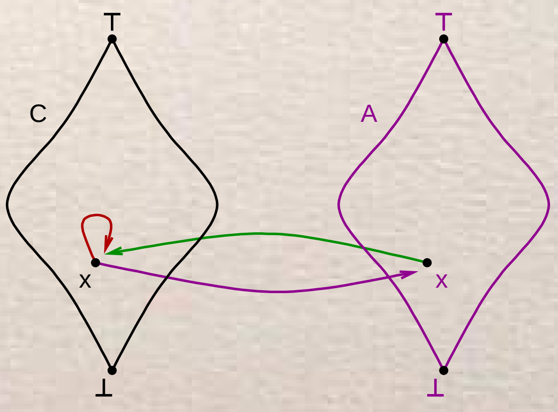
$< \mathrm{uco}(C), \sqsubseteq, \sqcap, \sqcup, \lambda x.\, \top, \lambda x.\, x >$

$$A_1 \sqsubseteq A_2 \quad \Leftrightarrow \quad A_2 \subseteq A_1$$
$$\sqcap_i A_i = \mathcal{M}(\cup_i A_i)$$
$$\sqcup_i A_i = \cap_i A_i$$

Top:



Bottom:



DEPENDENCIES IN PDG

Program Dependency Graphs (PDG) are a standard way for modelling dependencies for slicing. They are defined by two kind of edges (s_1, s_2):

CONTROL FLOW EDGE: s_1 represents a control predicate and s_2 represents a component of the program immediately nested within the predicate s_1;

FLOW DEPENDENCE EDGE: s_1 defines a variable x which is used in s_2
i.e., $x \in \mathbf{def}(s_1) \cap \mathbf{ref}(s_2)$,
and x is not further defined between s_1 and s_2;

DEPENDENCIES IN PDG

Program Dependency Graphs (PDG) are a standard way for modelling dependencies for slicing. They are defined by two kind of edges (s_1, s_2):

CONTROL FLOW EDGE: s_1 represents a control predicate and s_2 represents a component of the program immediately nested within the predicate s_1;

FLOW DEPENDENCE EDGE: s_1 defines a variable x which is used in s_2 i.e., $x \in \mathbf{def}(s_1) \cap \mathbf{ref}(s_2)$, and x is not further defined between s_1 and s_2;

$\Downarrow$

Flow dependence edges = DIRECT FLOWS=DEF-REF dependencies

Control flow edges = INDIRECT FLOWS

SLICING VS DEPENDENCIES

- SLICING $\Rightarrow$ Requires the same I/O behaviour, i.e., no *semantic* dependencies
- PDG $\Rightarrow$ Models *syntactic* dependencies

Slicing vs dependencies

- Slicing $\Rightarrow$ Requires the same I/O behaviour, i.e., no *semantic* dependencies
- Pdg $\Rightarrow$ Models *syntactic* dependencies

$$\Downarrow$$

There is a clear gap: Semantics vs Syntax

SLICING VS DEPENDENCIES

- SLICING $\Rightarrow$ Requires the same I/O behaviour, i.e., no *semantic* dependencies
- PDG $\Rightarrow$ Models *syntactic* dependencies

$$\Downarrow$$

THERE IS A CLEAR GAP: SEMANTICS VS SYNTAX

PDG $\Rightarrow$ Generate a slicing considering more dependencies (syntactic)

(SEMANTIC) SLICING $\Leftarrow$ Needs a *weaker* notion of dependence.

SLICING PARAMENTRIC ON THE CHOSEN DEPENDENCE NOTION!

A LOGIC FOR (IN)DEPENDENCIES

Formalization of notion of (in)dependence $[x \ltimes y]$[Amtoft & Banerjee '04]:

$G \vdash \{T_0^\#\}\ x := e\ \{T^\#\}$

if $\forall [y \ltimes w] \in T^\#.\ (x \neq y \;\Rightarrow\; [y \ltimes w] \in T_0^\#)$

$(x = y \;\Rightarrow\; (w \notin G \wedge \forall z \in FV(e).\ [z \ltimes w] \in T_0^\#)$

$$\frac{G_0 \vdash \{T_0^\#\} s_1 \{T^\#\} \quad G_0 \vdash \{T_0^\#\} s_2 \{T^\#\}}{G \vdash \{T_0^\#\} \textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2 \{T^\#\}}$$

if $G \subseteq G_0 \;\wedge\; (w \notin G_0 \;\Rightarrow\; \forall x \in FV(e).\ [x \ltimes w] \in T_0^\#)$

$$\frac{G_0 \vdash \{T^\#\} s \{T^\#\}}{G \vdash \{T^\#\} \textbf{while } e \textbf{ do } s \{T^\#\}}$$

if $G \subseteq G_0 \;\wedge\; (w \notin G_0 \;\Rightarrow\; \forall x \in FV(e).\ [x \ltimes w] \in T^\#)$

A LOGIC FOR (IN)DEPENDENCIES

Formalization of notion of (in)dependence $[x \ltimes y]$[Amtoft & Banerjee '04]:

$$G \vdash \{T_0^\#\}\ x := e\ \{T^\#\}$$

$$\text{if } \forall [y \ltimes w] \in T^\#.\ (x \neq y \ \Rightarrow\ [y \ltimes w] \in T_0^\#)$$

$$(x = y \ \Rightarrow\ (w \notin G \wedge\ \forall z \rightsquigarrow e. \qquad [z \ltimes w] \in T_0^\#)$$

$$\frac{G_0 \vdash \{T_0^\#\}s_1\{T^\#\} \quad G_0 \vdash \{T_0^\#\}s_2\{T^\#\}}{G \vdash \{T_0^\#\}\textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2\{T^\#\}}$$

$$\text{if } G \subseteq G_0\ \wedge\ (w \notin G_0\ \Rightarrow\ \forall x \rightsquigarrow e. \qquad [x \ltimes w] \in T_0^\#)$$

$$\frac{G_0 \vdash \{T^\#\}s\{T^\#\}}{G \vdash \{T^\#\}\textbf{while } e \textbf{ do } s\{T^\#\}}$$

$$\text{if } G \subseteq G_0\ \wedge\ (w \notin G_0\ \Rightarrow\ \forall x \rightsquigarrow e. \qquad [x \ltimes w] \in T^\#)$$

SEMANTIC (ABSTRACT) DEPENDENCIES

$$\boxed{z := w + y + 2x^2 - w}$$

SYNTACTIC DEP. A variable is a free variable in the expression assigned to z?

$\Longrightarrow$ z depends on $\{w, y, x\}$!

SEMANTIC (ABSTRACT) DEPENDENCIES

$$\boxed{z := w + y + 2x^2 - w}$$

SEMANTIC DEP. By varying the *value* of a variable does the expression change?

$$\boxed{x \leadsto_S e \;\Leftrightarrow\; \exists \sigma_1, \sigma_2.\, \forall y \neq x. \sigma_1(y) = \sigma_2(y) \;\wedge\; [\![e]\!](\sigma_1) = [\![e]\!](\sigma_2)}$$

$\Longrightarrow$ z depends on $\{y, x\}$!

SEMANTIC (ABSTRACT) DEPENDENCIES

$$\boxed{z := w + y + 2x^2 - w}$$

ABSTRACT SEMANTIC DEP. By varying the *property* of a variable does the *property* of the expression change?

$$\boxed{x \rightsquigarrow_{\mathsf{N}} e \;\Leftrightarrow\; \exists \sigma_1, \sigma_2. \forall y \neq x. \rho(\sigma_1(y)) = \rho(\sigma_2(y)) \;\wedge\; \rho([\![e]\!](\sigma_1)) = \rho([\![e]\!](\sigma_2))}$$

$\Longrightarrow$ If we consider *Parity* z depends on $\{y\}$!

SEMANTIC (ABSTRACT) DEPENDENCIES

$$\boxed{z := w + y + 2x^2 - w}$$

ABSTRACT SEMANTIC DEP. By varying the *property* of a variable does the *property* of the expression change?

$$\boxed{x \rightsquigarrow_{\mathsf{N}} e \;\Leftrightarrow\; \exists \sigma_1, \sigma_2.\, \forall y \neq x. \rho(\sigma_1(y)) = \rho(\sigma_2(y)) \;\wedge\; \rho([\![e]\!](\sigma_1)) = \rho([\![e]\!](\sigma_2))}$$

$\Longrightarrow$ If we consider *Sign* z depends on $\{y, x\}$!

PRUNING DEPENDENCIES

We have two kind of dependencies:

- Data dependencies (Assignments);
- Control dependencies (Control structures)

PRUNING DEPENDENCIES

We have two kind of dependencies:

- Data dependencies (Assignments) $\Rightarrow$ Direct flows;
- Control dependencies (Control structures) $\Rightarrow$ Indirect flows

Pruning dependencies

We have two kind of dependencies:

- Data dependencies (Assignments) $\Rightarrow$ Direct flows;
- Control dependencies (Control structures) $\Rightarrow$ Indirect flows

$$\boxed{\text{We propose a PRUNING of } \textit{data dependencies}!}$$

$\Downarrow$

STILL WE LOSE SOMETHING ABOUT CONTROL DEPENDENCIES!

Pruning dependencies

We have two kind of dependencies:

- Data dependencies (Assignments) $\Rightarrow$ Direct flows;
- Control dependencies (Control structures) $\Rightarrow$ Indirect flows

$$\textbf{if } (y + 2x \; mod \; 2) == 0 \textbf{ then } w := 0 \textbf{ else } w := 0$$

$\Rightarrow$ The guard DOES NOT DEPEND on x: OK

$\Rightarrow$ The variable w DOES NOT DEPEND on y: NO!

DERIVING ABSTRACT DEPENDENCIES (1)

The definition of narrow deps contains quantifiers on variables and states

This means that, even abstracting states,
the number of comparisons between $\rho\left(\llbracket e \rrbracket \sigma_1\right)$ and $\rho\left(\llbracket e \rrbracket \sigma_2\right)$
may be huge or infinite if the domain is non-trivial

Yet, we observe that:

- Some states are not *possible* at a given program point
- What is computed in a *broader* state can be valid in *narrower* states (monotonicity)

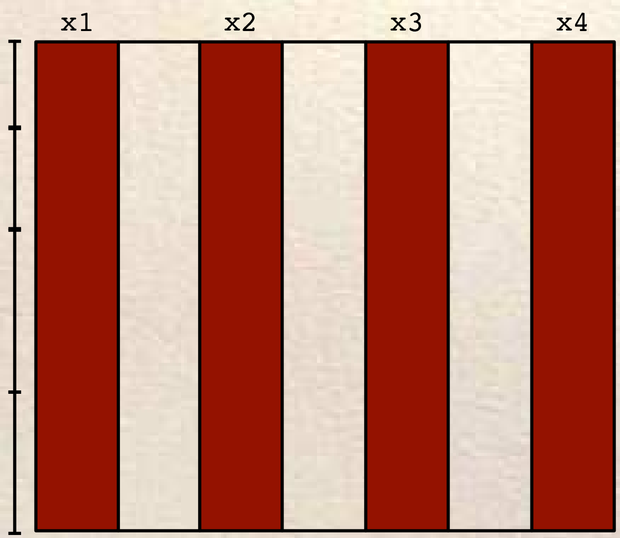
$$\rho(\llbracket e \rrbracket \langle [\top] \rangle) \leq \mathsf{U} \;\Leftrightarrow\; \rho(\llbracket e \rrbracket \langle [\mathbf{even}] \rangle) \leq \mathsf{U} \wedge \rho(\llbracket e \rrbracket \langle [\mathbf{odd}] \rangle) \leq \mathsf{U}$$

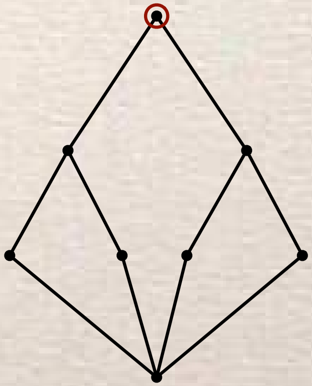
DERIVING ABSTRACT DEPENDENCIES (2)

- A systematic way to go through the (variables and states)-space:
  - incrementally find the set X of variables which are enough to determine the value of e
  - X determines e if any change to other variables can be ignored (needs to go into the state space)

DERIVING ABSTRACT DEPENDENCIES (2)

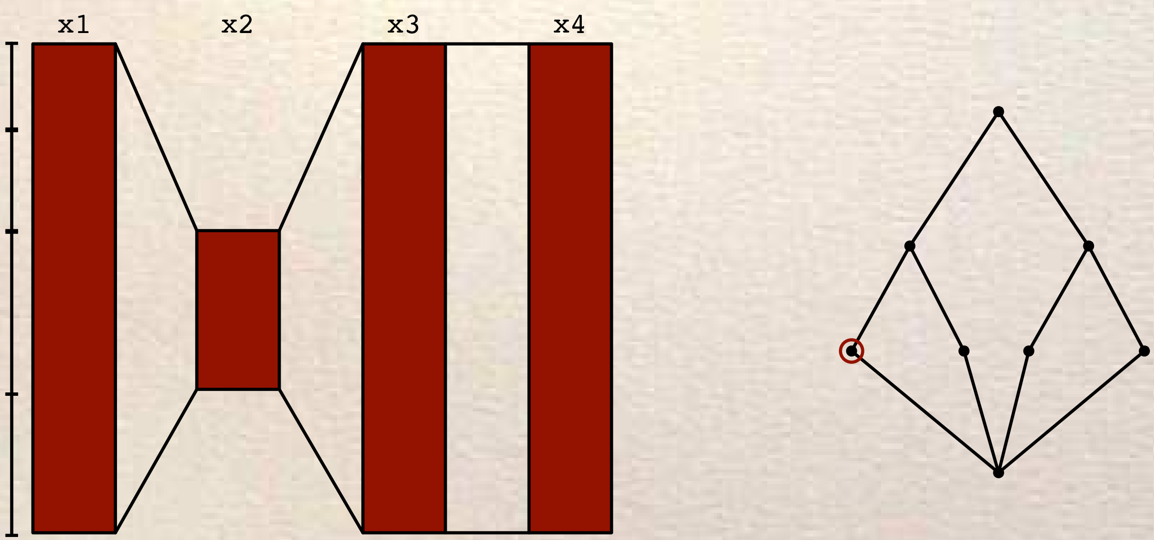
- A systematic way to go through the (variables and states)-space:
  - incrementally find the set X of variables which are enough to determine the value of e
  - X determines e if any change to other variables can be ignored (needs to go into the state space)





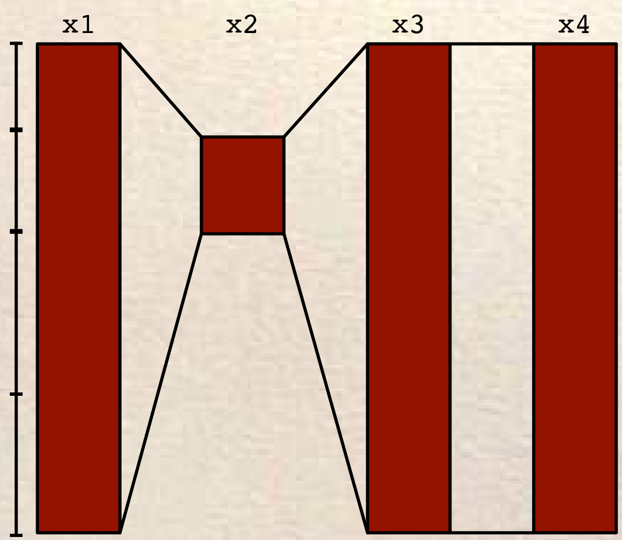
Deriving abstract dependencies (2)

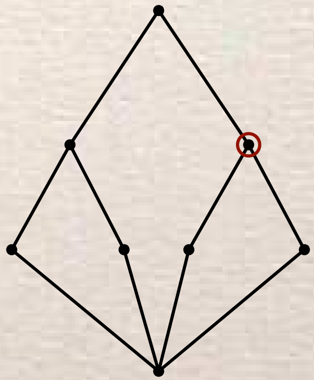
- A systematic way to go through the (variables and states)-space:
  - incrementally find the set X of variables which are enough to determine the value of e
  - X determines e if any change to other variables can be ignored (needs to go into the state space)



DERIVING ABSTRACT DEPENDENCIES (2)

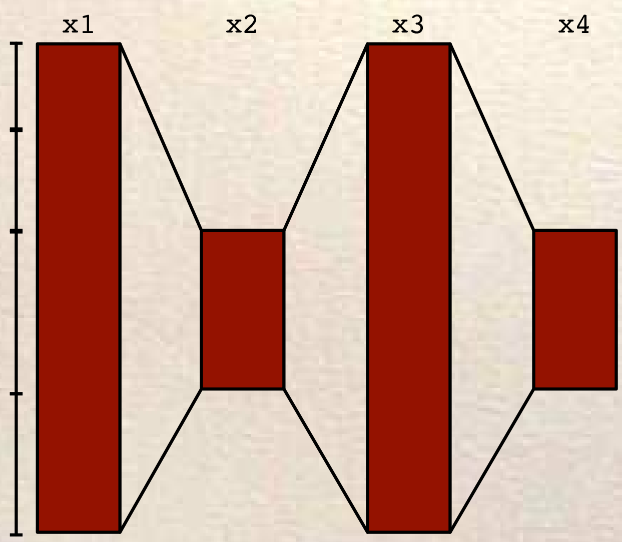
- A systematic way to go through the (variables and states)-space:
  - incrementally find the set X of variables which are enough to determine the value of e
  - X determines e if any change to other variables can be ignored (needs to go into the state space)

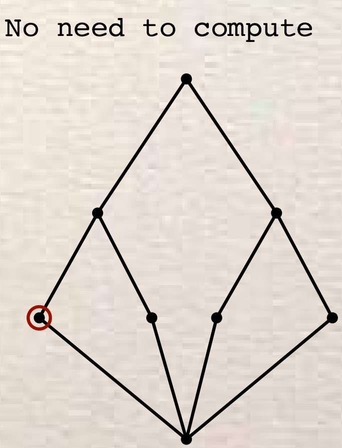




DERIVING ABSTRACT DEPENDENCIES (2)

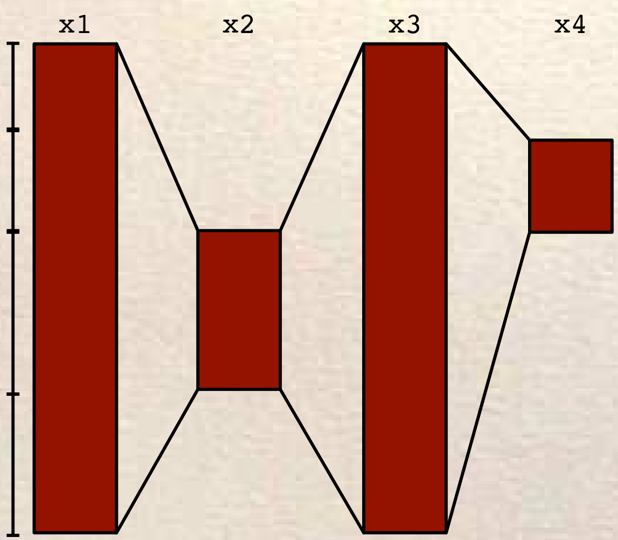
- A systematic way to go through the (variables and states)-space:
  - incrementally find the set X of variables which are enough to determine the value of e
  - X determines e if any change to other variables can be ignored (needs to go into the state space)

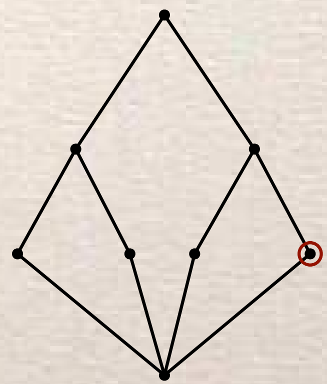




DERIVING ABSTRACT DEPENDENCIES (2)

- A systematic way to go through the (variables and states)-space:
  - incrementally find the set X of variables which are enough to determine the value of e
  - X determines e if any change to other variables can be ignored (needs to go into the state space)

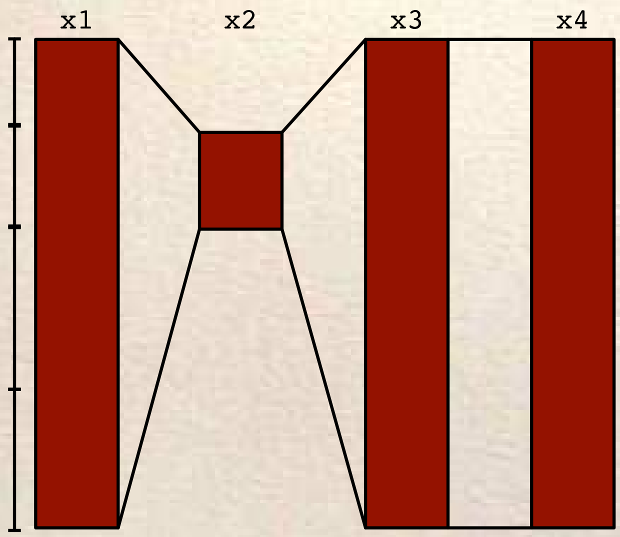


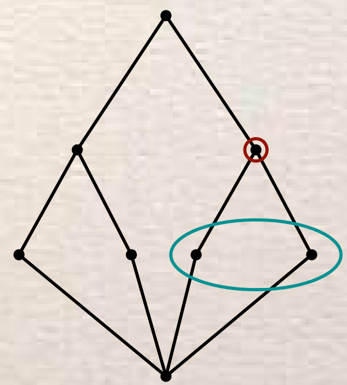


REMOVING ABSTRACT DEPENDENCIES

Another application: simplifying a domain in order to remove dependencies on a set of variables

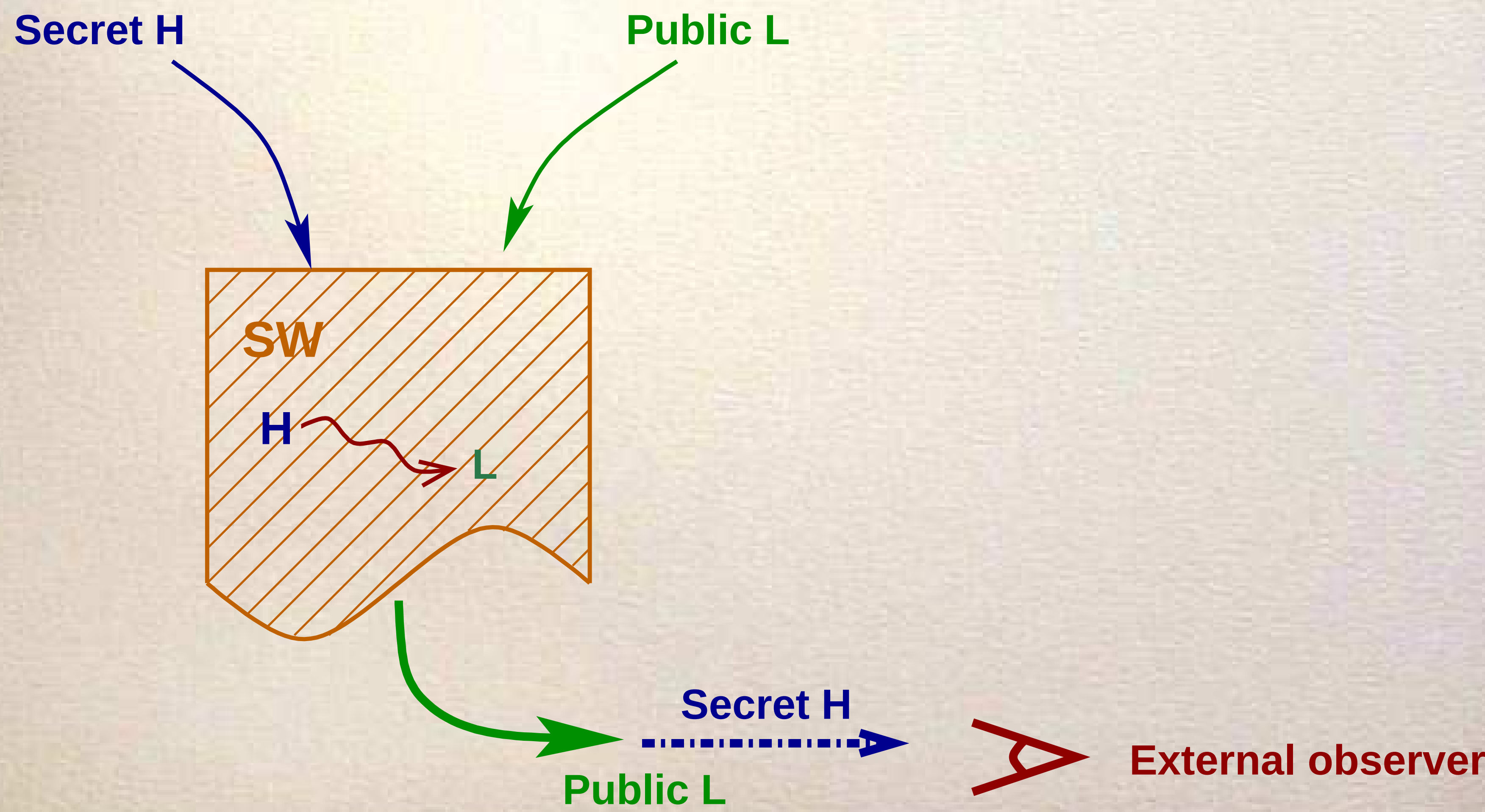
- Basically, systematically removing from ρ the abstract values which are responsible for the distinguishability of two states





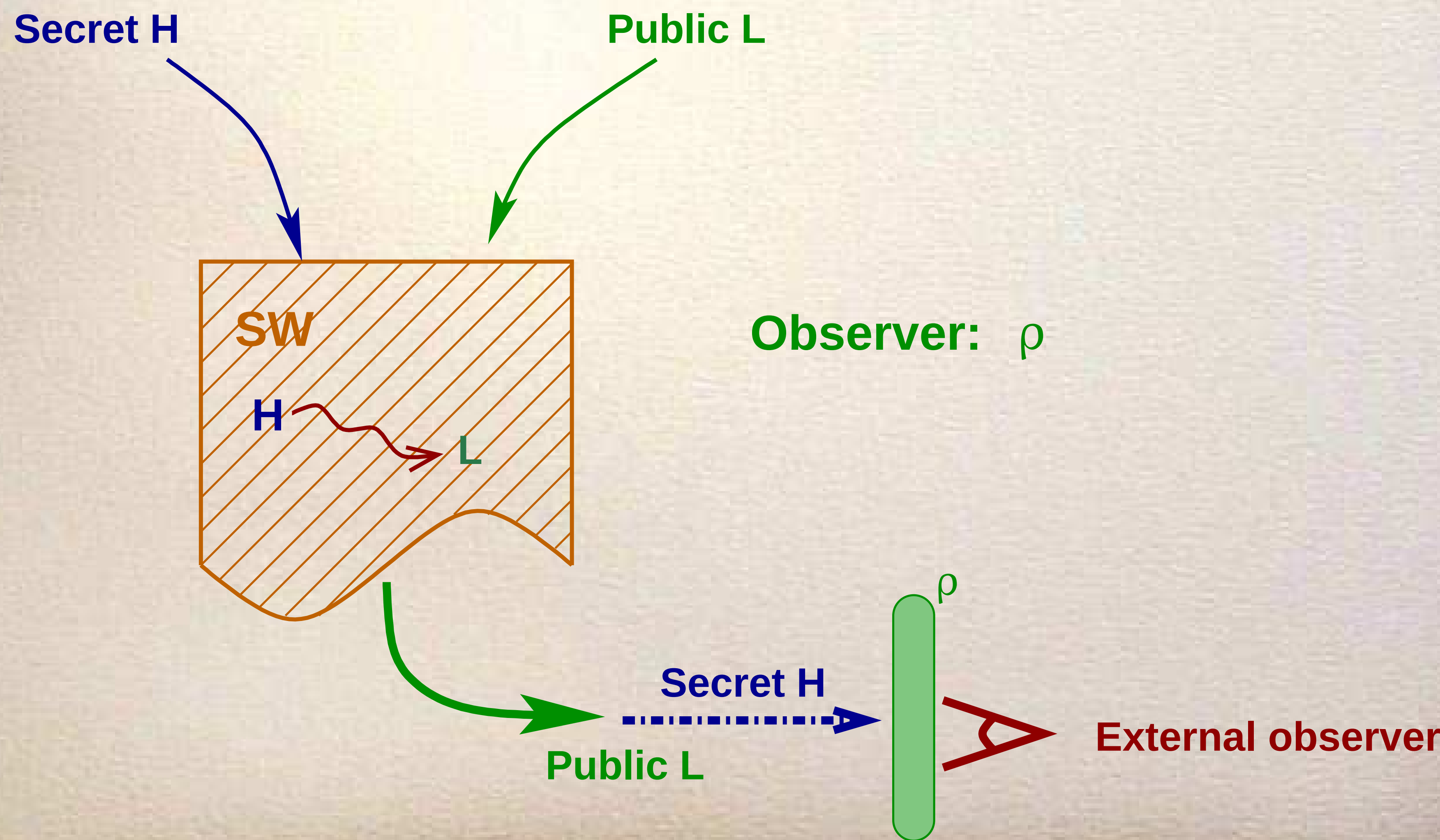
ABSTRACT NON-INTERFERENCE

[Giacobazzi & Mastroeni ’04]



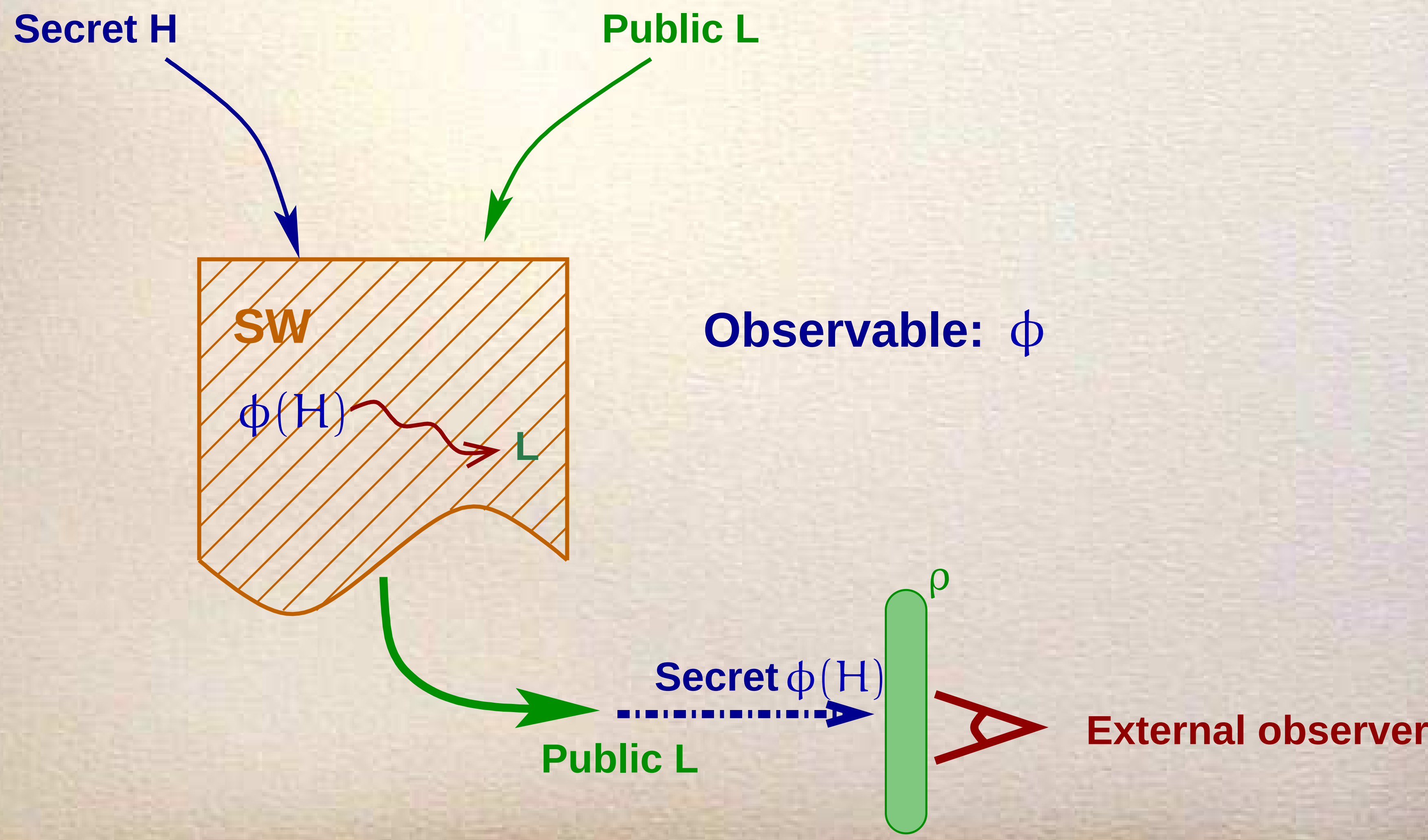
ABSTRACT NON-INTERFERENCE

[Giacobazzi & Mastroeni '04]



ABSTRACT NON-INTERFERENCE

[Giacobazzi & Mastroeni ’04]



CERTIFYING PROGRAMS FOR ANI

- We certify the security degree of programs relatively to an output observation [Giacobazzi & Mastroeni].
- We can derive the certification inductively on the *syntax* of programs [Giacobazzi & Mastroeni '04].
- PROBLEM: This system of rules has a *semantic rule!*

CERTIFYING PROGRAMS FOR ANI

- We certify the security degree of programs relatively to an output observation [Giacobazzi & Mastroeni].
- We can derive the certification inductively on the *syntax* of programs [Giacobazzi & Mastroeni '04].
- PROBLEM: This system of rules has a *semantic rule!*

We can avoid the semantic rule by computing ABSTRACT DEPENDENCIES for assignment!

CONCLUSIONS

- We provide an insight on the strong relation between slicing and dependency;
- A new point of view: Slicing parametric on a notion of dependency;
- Still we are not able to get the most precise semantic slicing;
- Still there is a lot of work to do towards a real implementation.