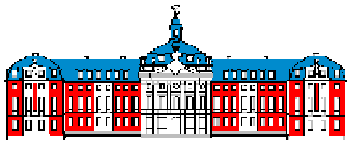


Information Flow and Program Analysis

Markus Müller-Olm
Westfälische Wilhelms-Universität Münster, Germany

IFIP WG 2.2 Meeting
Singapore, September 13-16, 2016

Project Context

Work in progress from a joint project with G. Snelting (KIT)

- Information flow control for mobile components based on precise analysis of paralle programs

Part of priority programme 1496

- Reliably Secure Software Systems (RS3)

funded by DFG (German Research Foundation)

Special thanks to Benedikt Nordhoff

What This Talk is About

Theme:

How can program analysis-like technology help PDG-based information flow analysis?

Program analysis:

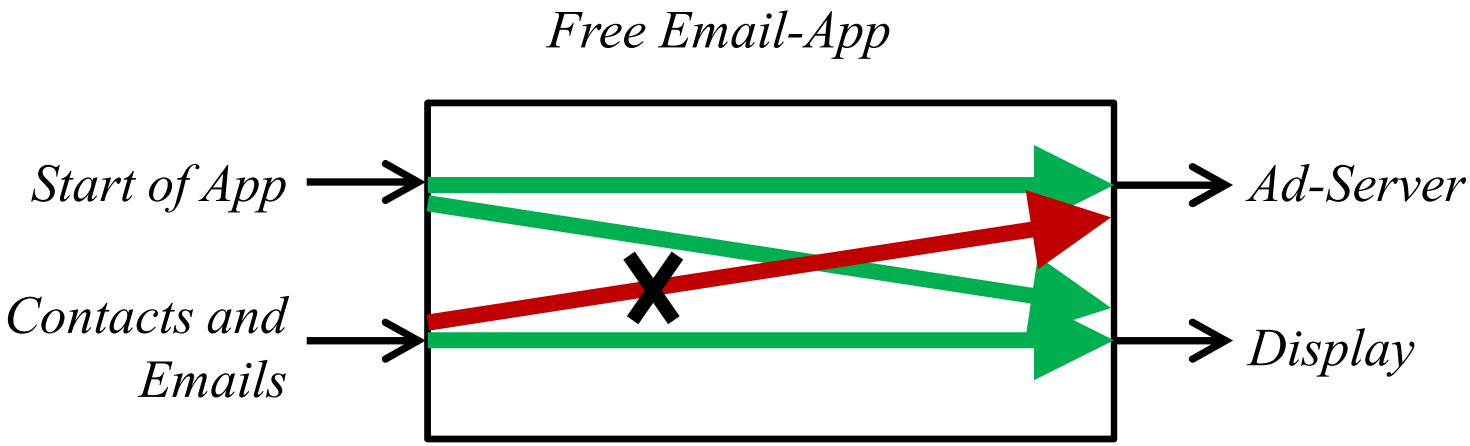
Fixpoint-based methods:

data-flow analysis, abstract interpretation

Information flow analysis:

see next slide

Information Flow: Example



Reference scenario of SPP RS3:

- Software security for mobile devices
- Prototype of certifying app store for Android (Lortz et. al., ...)

Non-Interference

For simplicity: transformational terminating programs only

Semantic setup:

Variables:	$Var = Low \cup High$
States:	$\Sigma = \{ \sigma \mid \sigma : Var \rightarrow Val \}$
Program semantics:	$[\![\pi]\!] : \Sigma \rightarrow \Sigma$

Low-equivalence of states:

$$\sigma \sim_L \sigma' \quad :\Leftrightarrow \quad \sigma|_{Low} = \sigma'|_{Low}$$

Program π is called non-interferent iff f.a. $\sigma,\sigma' \in \Sigma$:

$$\sigma \sim_L \sigma' \quad \Rightarrow \quad [\![\pi]\!](\sigma) \sim_L [\![\pi]\!](\sigma')$$

Possibilistic Non-Interference

Semantics of non-deterministic programs:

$$[\![\pi]\!] : \Sigma \rightarrow 2^{\Sigma}$$

Refinement:

$$\pi \sqsubseteq \pi' \quad \Leftrightarrow \qquad \forall \sigma: \quad [\![\pi]\!](\sigma) \subseteq [\![\pi]\!](\sigma')$$

Program π is called non-interferent iff f.a. $\sigma_1, \sigma_2 \in \Sigma$:

$$\sigma \sim_L \sigma' \;\Rightarrow\; \forall \rho \in [\![\pi]\!](\sigma) : \exists \rho' \in [\![\pi]\!](\sigma') \;:\; \rho \sim_L \rho'$$

Observation: Non-interference is not preserved by refinement.

Example: l := ? is non-interferent, its refinement l := h is not

Reason: Non-interference is a „hyper-property“

A Fundamental Problem

- Abstraction is inherent to program analysis
- However, as just observed:

  Non-interference does not transfer from abstractions
- Consequence:

  Program analysis cannot be directly applied to non-interference

Program Dependence Graphs (PDGs)

- A structure known from program slicing
- Nodes correspond to statements and conditions;
  we add artificial nodes for initial and final value of program variables
- Edges capture data dependences and control dependences
- PDGs can be applied for non-interference analysis

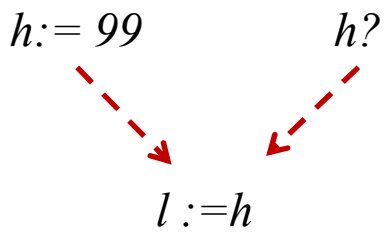
  Analysis principle:

  If there is no path in PDG from high input to low output
  then the program is non-interferent

Direct and Indirect Flows

Direct flows:

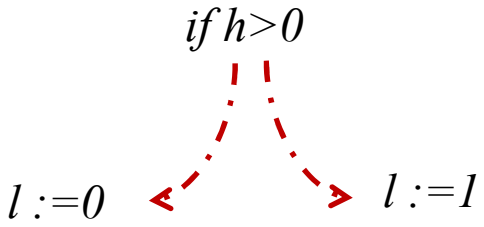
l := h



captured by data dependence edges in PDG

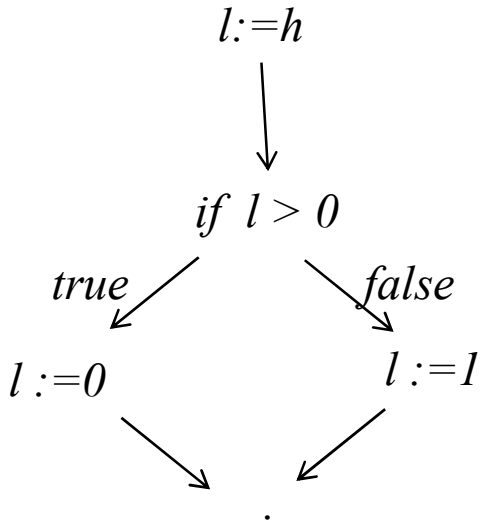
Indirect flows:

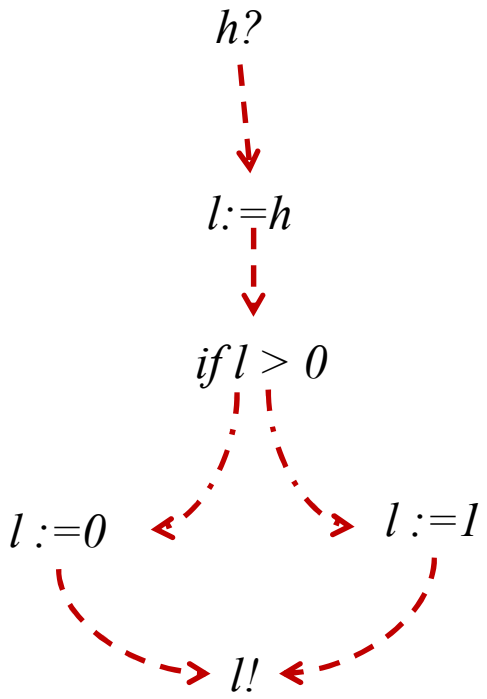
if h>0 then l := 0 else l := 1



captured by control dependence edges in PDG

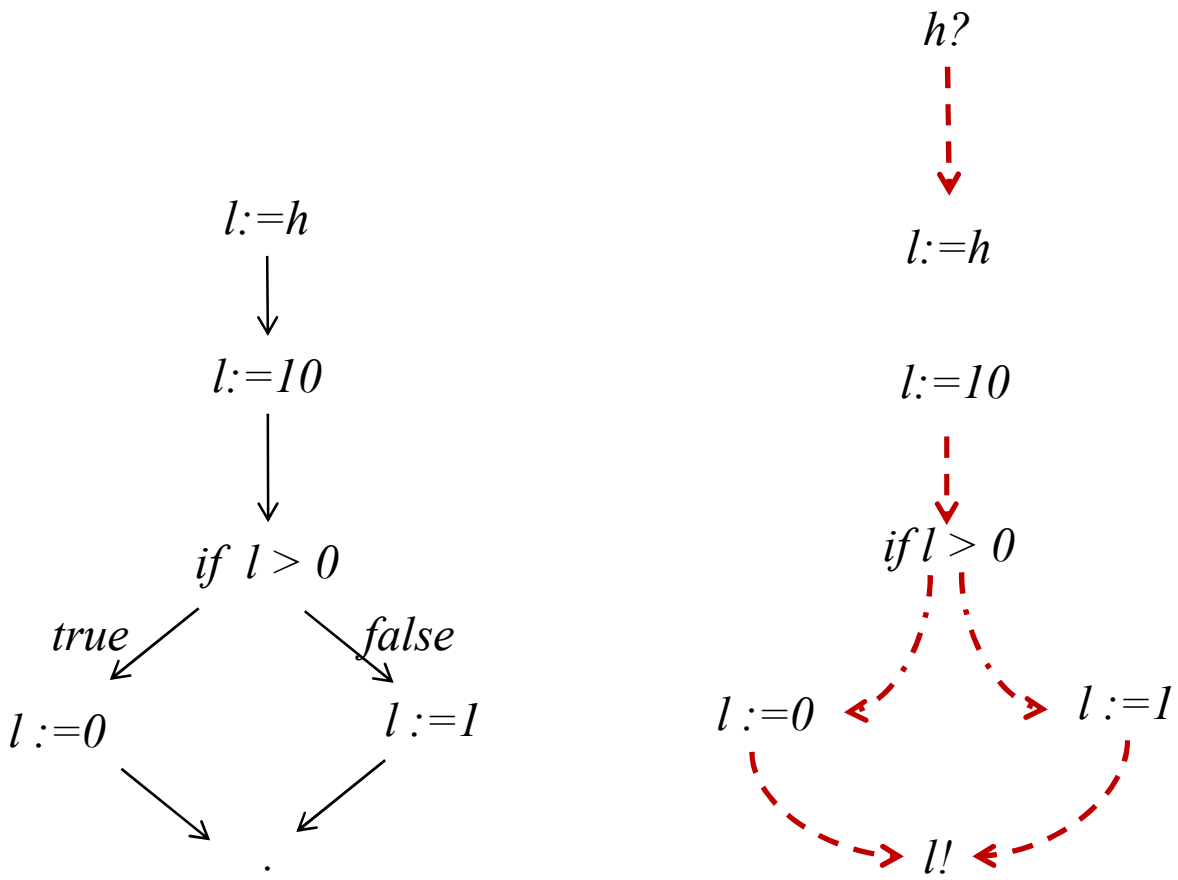
Example 1





There is a path from h? to l!. Hence: Program may be interferent

Example 2



There is no path from h? to l!. Hence: Program is non-interferent

Path Conditions

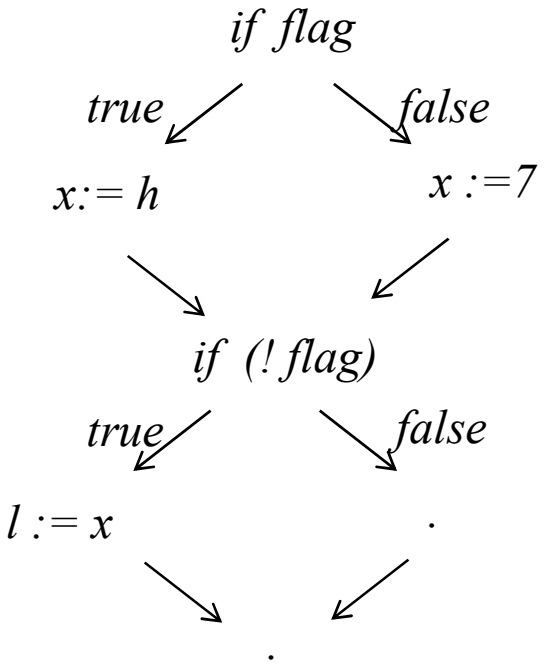
[Snelting]

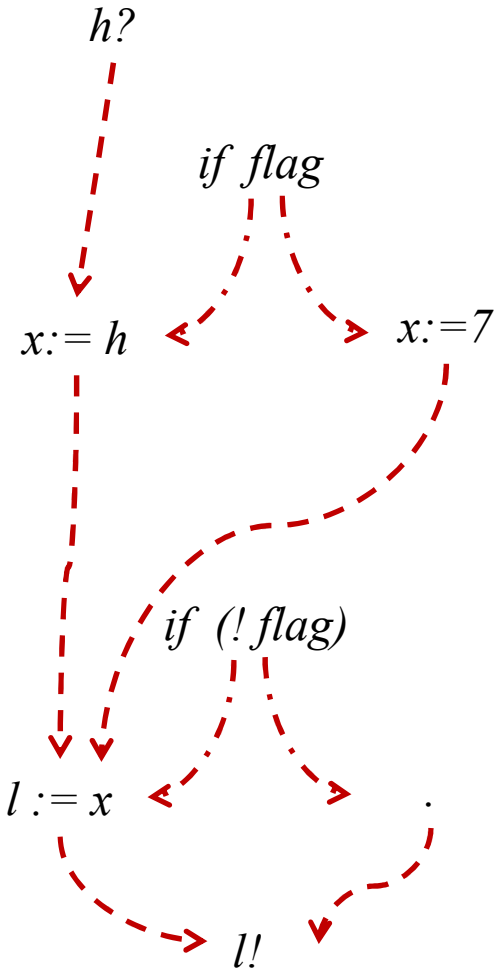
Goal: Improve precision of PDG-based dependence analysis

Idea: For each path in the PDG indicating critical flow, read off a necessary condition for flow from the guards. If all these conditinos are unsatisfiable, there is no flow.

Caveat: Requires SSA-form of programs

Path conditions improve precision of PDGs



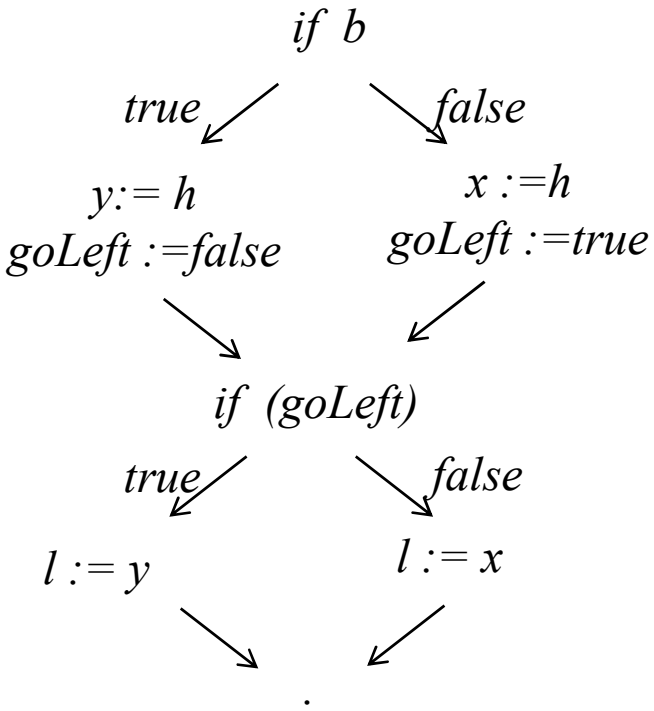


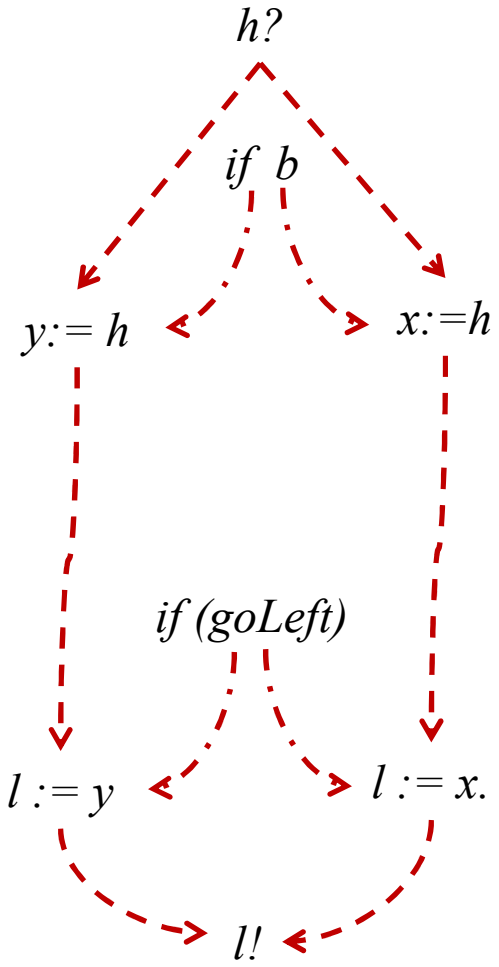
PDG alone: false alarm

+ path conditions: OK

flag ∧ *! flag*

Further Improvements by Data Analysis Desirable





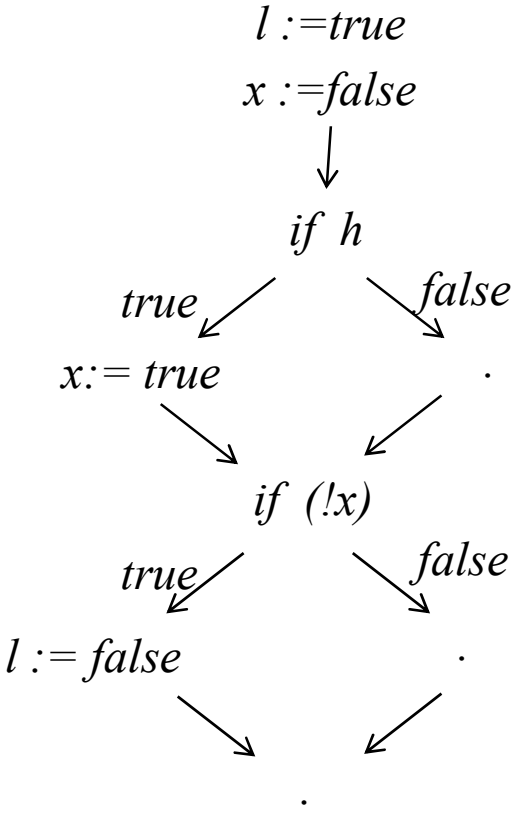
PDG + path conditions: false alarm

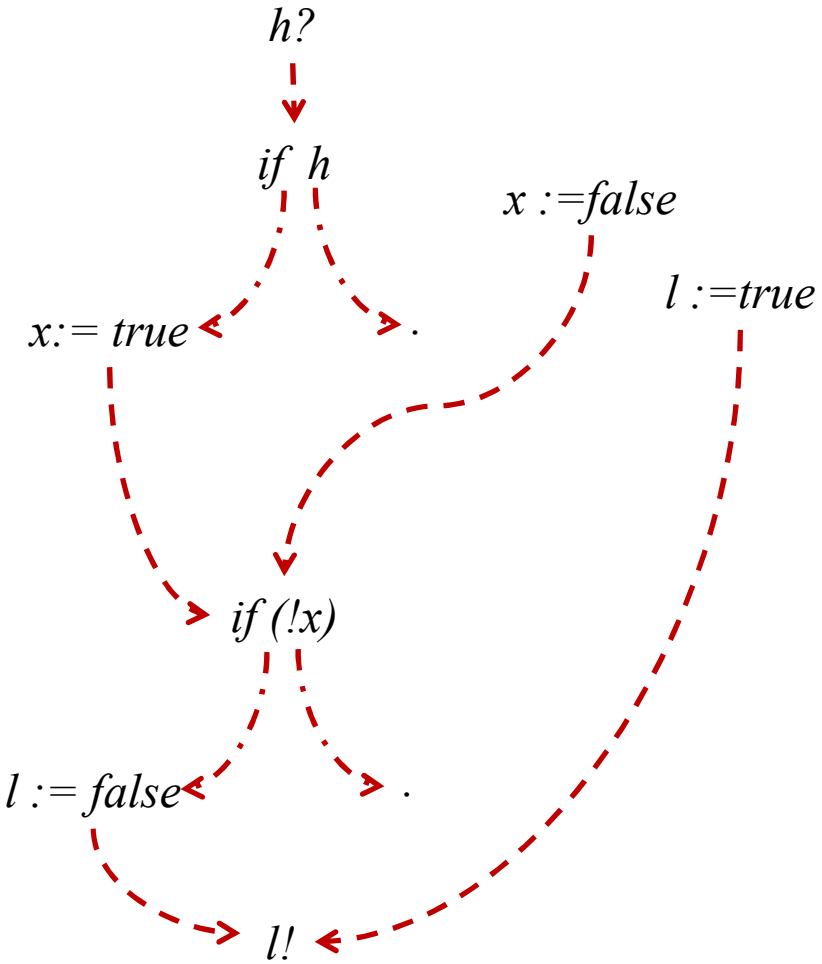
+ invariant: OK

For left path: $b \wedge \ goLeft \wedge goLeft = !\, b$

For right path: $!\, b \wedge \ !\, goLeft \wedge goLeft = !\, b$

The Show Stopper





PDG + path conditions

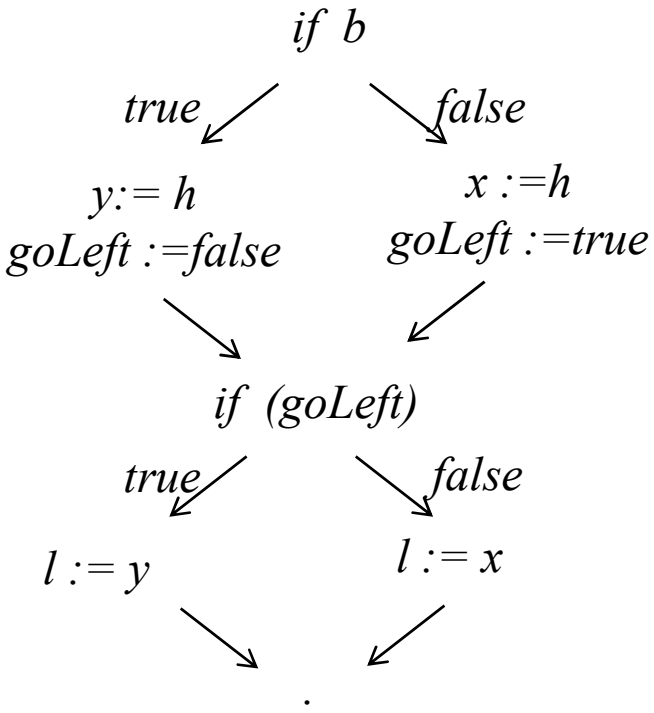
+ invariant: unsound

$h \wedge !x \wedge x = h$

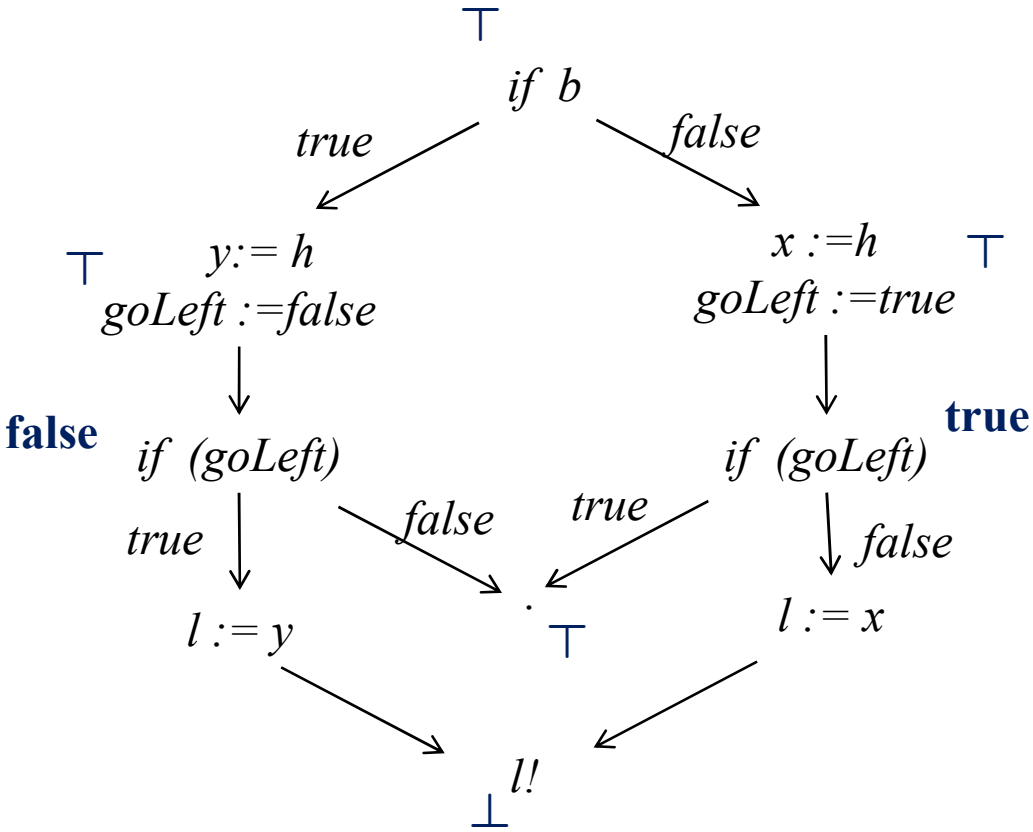
A Glimpse on Data Flow Slicing

- Guiding intuition: Flow happens along PDG paths only
- Add new type of dependencies (data control dependencies) to avoid soundness problem
- Define executions along a PDG path
- Prove: If program has no execution along a critical PDG path, then program is non-interferent (Isabelle!)
- Actual analysis
  - Generate a program whose executions correspond exactly to the executions along critical PDG paths
  - Check by data flow analysis/abstract interpretation whether final control point is reachable
- Note: Approach allows to check non-interference by safety analysis !

A Glimpse on Data Flow Slicing: Example



Original program



Generated program

Constant propagation on the generated program proves absence of critical information flow

Discussion

Further work:

- Use DPNs to help PDG-based non-interference analysis of parallel programs based on LSOD
- Use DPNs to help type-based non-interference analysis of parallel programs

Alternative approaches:

- Self-composition
- Hyper-logics

Certifying App Store

Thank you !