Bidirectional Data Flow Analysis : Past Beliefs and Recent Advances

Uday Khedker

(uday@cs.unipune.ernet.in)

Department of Computer Science

University of Pune, India

Outline

- Data Flow Analysis
- Bidirectionality in Data Flow Analysis
- Myths in Bidirectional Data Flow Analysis
- The Realities
- Limitations of Formulating Bidirectional Flows as Unidirectional Flows
- Recent Advances
  - Bidirectional Dependencies in Type Inferencing
  - Data Flow Analysis as a Special Case of General Program Property Inference
  - More Powerful Dateness Analysis
- Conclusions

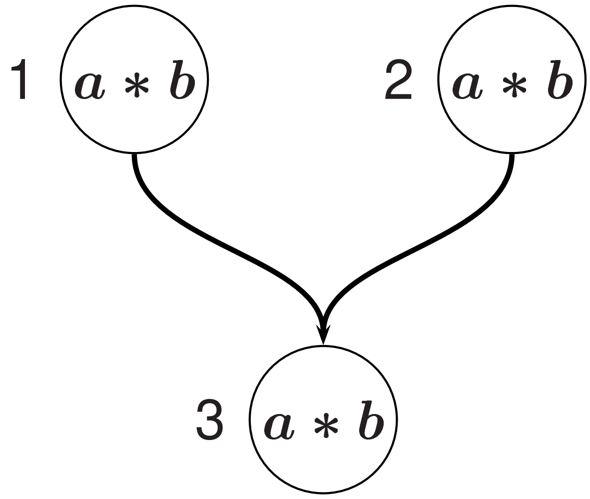
Data Flow Analysis

- Analysis to capture Semantic Information
  - Code Optimization
  - Code Development
    (Programming Environments)
  - Code re-engineering/migration/maintenance
- Performed over internal representation

  (Usually Control Flow Graphs)
- Examples :
  - Available Expressions Analysis
  - Reaching Definitions Analysis
  - Placement Possibility Analysis for Partial Redundancy Elimination (PRE)
  - ...

An Example of Data Flow Analysis

Objective : Common Subexpression Elimination



- Along paths $\mathbf{1} \rightarrow \mathbf{3}$, $\mathbf{2} \rightarrow \mathbf{3}$: a, b not modified.
- Along path $\mathbf{1} \rightarrow \mathbf{3}$:

  Val($a * b$) in node 3 $\equiv$ Val($a * b$) in node 1

  Along path $\mathbf{2} \rightarrow \mathbf{3}$:

  Val($a * b$) in node 3 $\equiv$ Val($a * b$) in node 2
- $a * b$ in 3 : Redundant recomputation.

  Previously computed value can be used.

Available Expressions Analysis

- Definition :

  An expression e is available at a program point u, if it *preserves* the value of its previous computation along *all* paths reaching u.

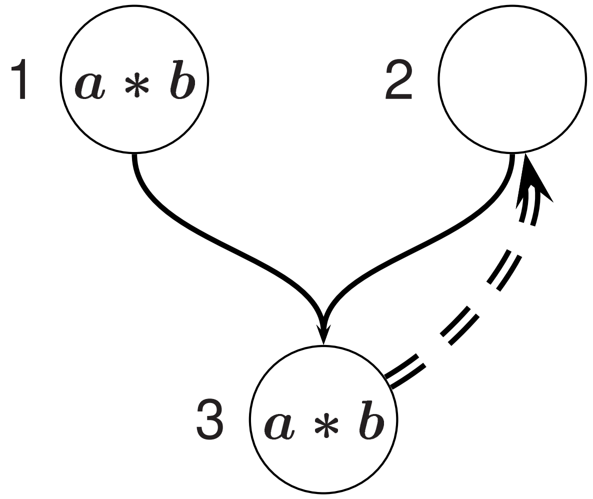
- Target :

  To identify the maximal set of expressions available at each program point.

- Use : Common Subexpressions Elimination

More Advanced Data Flow Analysis

Partial Redundancy Elimination (PRE)
(Morel & Renvoise, 1979)



- Computation of $a * b$ in node 3 is
  - redundant along path $\mathbf{1} \rightarrow \mathbf{3}$, but ...
  - *not* redundant along path $\mathbf{2} \rightarrow \mathbf{3}$.

- Insert $a * b$ in node 2.

  $\Rightarrow$ Computation of $a * b$ in node 3 is redundant along all paths.

  $\Rightarrow$ Can be eliminated.

Direction in Data Flow Analysis

Info. at a point depends upon the info. at

- (Control Flow) Predecessors

  $\Rightarrow$ Forward Analysis

- (Control Flow) Successors

  $\Rightarrow$ Backward Analysis

- Both Predecessors as well as Successors

  $\Rightarrow$ Bidirectional Analysis

Bidirectional Data Flows

- Combinations of Forward and Backward Flows

  Traditional Applications :

  – PRE and related optimizations
  – Code Hoisting and Strength Reduction
  – Live Range Characterisation and Register Assignment
  – Dead Code Elimination
  – Communication Placement
  – Shrink Wrapping of Procedure Calls
  – Redundant Array Bound Checks Elimination
  – Compilation for Distributed Memory

Types of Bidirectional Dependencies

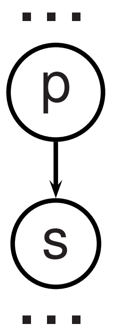
- Separate Flows

  Ex : Live Range Characterization

  (Most Common)

- Simultaneous Flows

  Ex : Traditional formulation of PRE

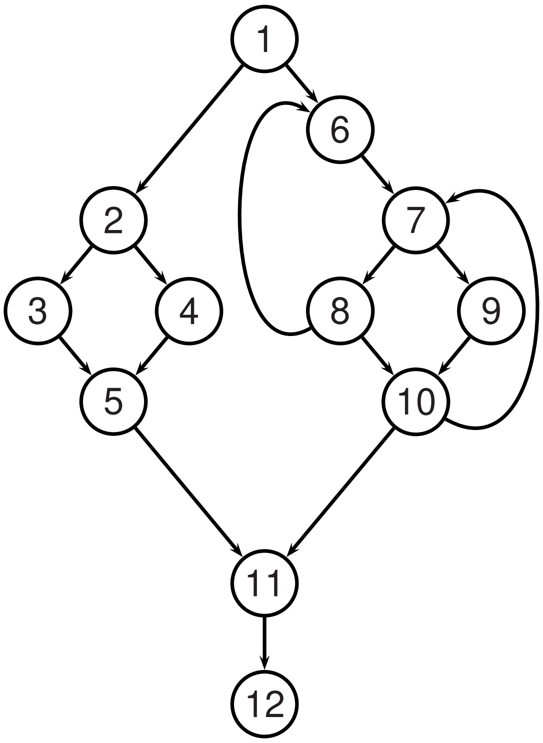


IN_s depends on OUT_p

OUT_p depends on IN_s

Why Should Bidirectionality Matter?

- Separate Flows : Bidirectionality does not matter.
- Simultaneous Flows : Data Flows may not follow graph theoretic paths.



Example :

$1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 11 \rightarrow 10 \rightarrow 7 \rightarrow 8 \rightarrow 6$

Why Should Bidirectionality Matter?

Data Flows may not follow graph theoretic paths.

- So What ?

  Practically,

  - Correctness $\surd$
  - Efficiency ??

  Theoretically,

  - Correctness ??
  - Complexity

    Computational ??

    Conceptual ??

Why Should Bidirectionality Matter?

Data Flows may not follow graph theoretic paths.

Practical observations

$\uparrow \quad \uparrow \quad \uparrow \quad \uparrow \quad \uparrow$

Massive Discrepancies

$\downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow$

Expected behaviour

- Round robin iterative analysis of PRE :

Complexity :	empirical	$- < 6$ iterations
	theoretical	– unknown.
	closest estimate	– $O(n^2)$!

- $\boxed{\text{Profound Mysteries} \Rightarrow \text{Profound Myths}}$

Myths in Bidirectional Data Flow Analysis

Strengthened by Knoop, Ruthing, et. al.

(Partially) Endorsed by Muchnick (1997)

- Bidirectional Flows :

  Conceptually and Computationally More Complex

- Bidirectional Flows Should be avoided

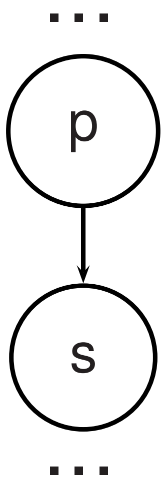
- Should be converted into Unidirectional Flows
  - Edge placement,
  - Edge splitting,
  - Decomposition into unidirectional flows

- Can be done so for *all* bidirectional flows.

  (Ruthing, 1998)

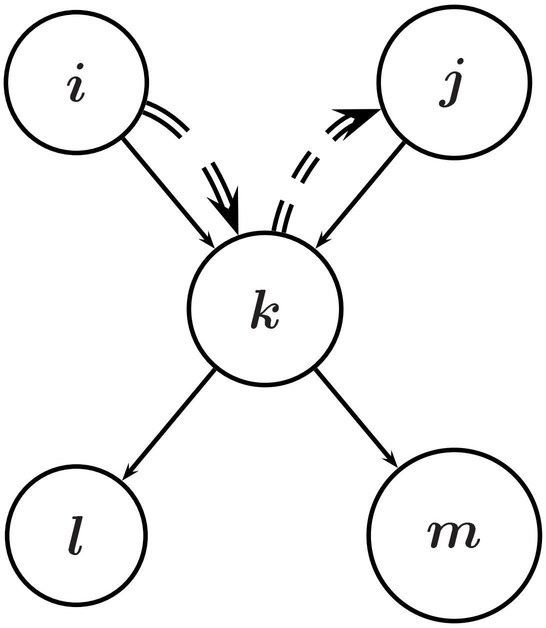
Conceptual Complexity of Bidirectional Flows

- Examining Bidirectional Dependencies



Information Flow	Indicated By
$IN_p \rightarrow OUT_p$	N_p^f
$OUT_p \rightarrow IN_p$	N_p^b
$OUT_p \rightarrow IN_s$	$E_{(p,s)}^f$
$IN_s \rightarrow OUT_p$	$E_{(p,s)}^b$

Conceptual Complexity of Bidirectional Flows



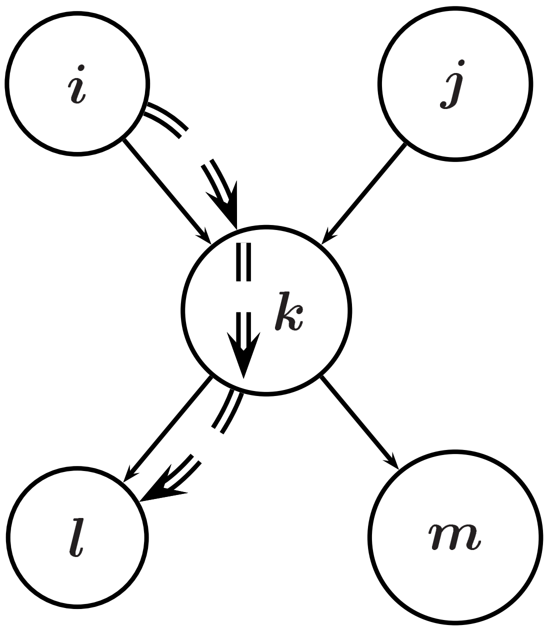
- Information Flow : $i \rightarrow k \rightarrow j$
- Can be realised by the traversals :

$$\text{OUT}_i \xrightarrow{E^f_{(i,k)}} \text{IN}_k \xrightarrow{E^b_{(j,k)}} \text{OUT}_j$$

- No flow *through* node k

  Traversals : Edge Traversals
  Direction : Alternate Forward and Backward

Conceptual Complexity of Bidirectional Flows



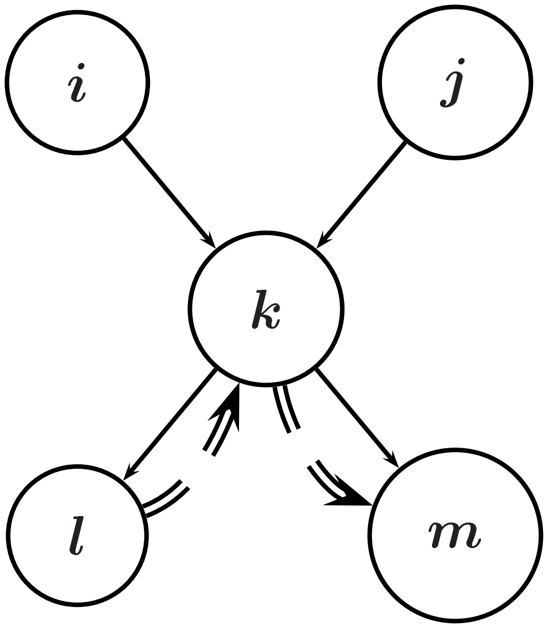
- Information Flow : $i \rightarrow k \rightarrow l$
- Can be realised by the traversals :

$$\text{OUT}_i \xrightarrow{E^f_{(i,k)}} \text{IN}_k \xrightarrow{N^f_k} \text{OUT}_k \xrightarrow{E^f_{(k,l)}} \text{IN}_l$$

- Forward Flow

  Traversals : Alternate Node and Edge Traversals
  Direction : Forward

Conceptual Complexity of Bidirectional Flows



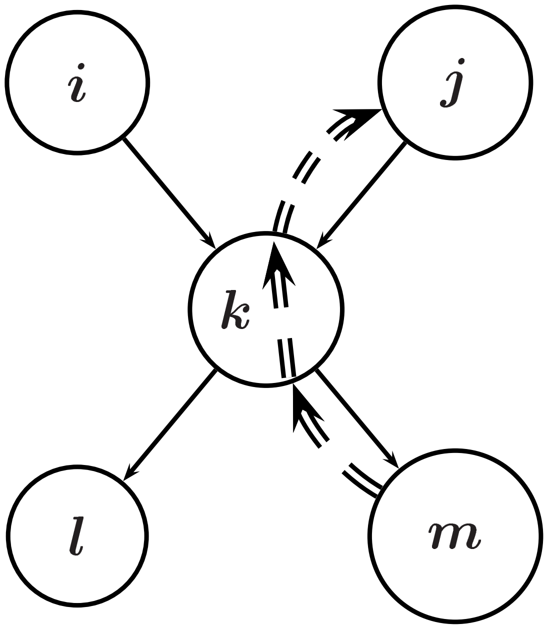
- Information Flow : $l \rightarrow k \rightarrow m$
- Can be realised by the traversals :

$$\text{IN}_l \xrightarrow{E^b_{(k,l)}} \text{OUT}_k \xrightarrow{E^f_{(k,m)}} \text{IN}_m$$

- No flow *through* node k

  Traversals : Edge Traversals
  Direction : Alternate Forward and Backward

Conceptual Complexity of Bidirectional Flows



- Information Flow : $m \rightarrow k \rightarrow j$
- Can be realised by the traversals :

$$\text{IN}_m \xrightarrow{E^b_{(k,m)}} \text{OUT}_k \xrightarrow{N^b_k} \text{IN}_k \xrightarrow{E^b_{(j,k)}} \text{OUT}_i$$

- Backward Flow

  Traversals : Alternate Node and Edge Traversals
  Direction : Backward

Computational Complexity of Bidirectional Flows

- Number of Nodes : n

  Number of Expressions : r

- Worklist based methods

  $\boxed{\text{Same as Unidirectional Flows : } O(n \cdot r)}$

- Round Robin iterative method

  A little higher

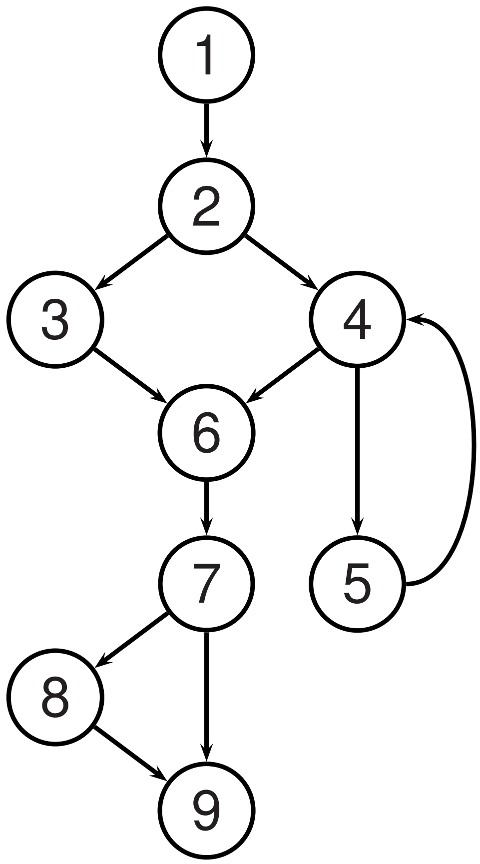
  Unidirectional : $O((d+1) \cdot n \cdot r)$
  d is the depth of the graph
  Bidirectional : $O((w+1) \cdot n \cdot r)$
  w is the width of the graph

  Earlier Estimate : $O(n^3 \cdot r)$

- Results in POPL 93 and TOPLAS 94.

Formulating Bidirectional Flows as Unidirectional Flows

A closer look at the semantics of PRE



- PPIN_3 depends on PPOUT_2
  but not on PPOUT_1.
- PPIN_3 depends on PPOUT of 6, 7, 8, and 9.

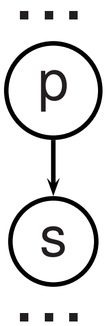
Formulating PRE Using Unidirectional Flows

- Forward dependencies in PRE are single-level

  Backward dependencies in PRE are multi-level

- Backward Node and Edge flows exist but . . .

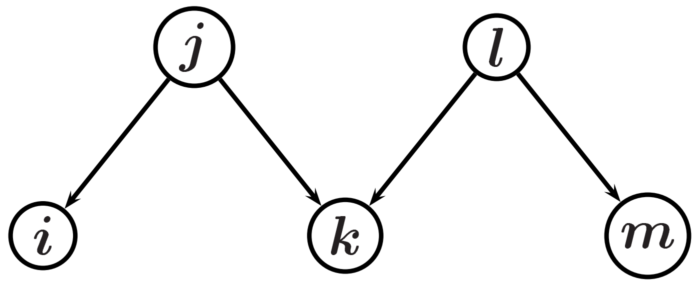
  Forward Node Flow does not exist



$$
\begin{array}{rcl}
\mathrm{IN}_{\boldsymbol{p}} \xrightarrow{N_p^f} \mathrm{OUT}_{\boldsymbol{p}} & : & \times \\
\mathrm{OUT}_{\boldsymbol{p}} \xrightarrow{N_p^b} \mathrm{IN}_{\boldsymbol{p}} & : & \surd \\
\mathrm{OUT}_{\boldsymbol{p}} \xrightarrow{E^f_{(p,s)}} \mathrm{IN}_{\boldsymbol{s}} & : & \surd \\
\mathrm{IN}_{\boldsymbol{s}} \xrightarrow{E^b_{(s,p)}} \mathrm{OUT}_{\boldsymbol{p}} & : & \surd
\end{array}
$$

Formulating Bidirectional Flows as Unidirectional Flows

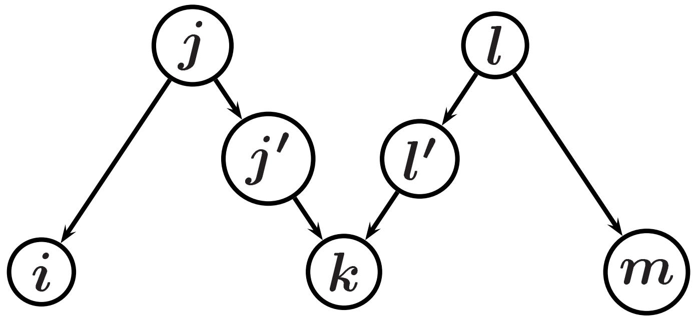
- Without Edge Placement/Splitting



$$\text{IN}_i \xrightarrow{E^b_{(i,j)}} \text{OUT}_j \xrightarrow{E^f_{(j,k)}} \text{IN}_k \xrightarrow{E^b_{(k,l)}} \text{OUT}_l \xrightarrow{E^f_{(l,m)}} \text{IN}_l$$

Formulating Bidirectional Flows as Unidirectional Flows

- With Edge Placement/Splitting



$\text{IN}_{j'} \xrightarrow{N^f_{j'}} \text{OUT}_{j'}$ does not exist.

$\Rightarrow$ Forward Flows are Restricted
(Only one forward edge after backward flow,
No backward flow after a forward flow)

$\Rightarrow$ Backward Data Flow Analysis
+
One forward correction pass
(Dhamdhere, SIGPLAN Notices 88)

Limitations of Formulating Bidirectional Flows as Unidirectional Flows

- Edge Placement/Splitting does not work for "Truly" Bidirectional Flows.
- Works only for PRE-like flows.

  However (even for these flows),

  - If Edge Placement/Splitting is not done
    $\Rightarrow$ Unidirectional formulation is not possible!
  - If Edge Placement/Splitting is done
    $\Rightarrow$ Alternative formulations are not required!

Recent Advances

- PRE is only partially bidirectional.
- Are there any useful "truly" bidirectional data flow analyses?
  - Past Belief : No.
    $\Rightarrow$ Bidirectional Data Flow Analysis :
    Much Ado About Nothing?
  - Recent Discovery : Yep!
    We saw them but didn't recognise them!
- Data Flow Analysis is a special case of program property inference mechanisms without auxiliary store.

Traditional Type Inferencing

- Type constraints introduced by uses of the variables.
- Statically Types Languages :

  Constraints have a global effect.
- Type Inferencing (Hindley-Milner Approach)
  - Identify constraints and store them in an auxiliary store
    $\Rightarrow$ One pass over the program
  - Use an auxiliary logic to solve constraints
    No sense of order between constraints
    $\Rightarrow$ No sense of "direction"

  Usually considered a single pass algorithm.

Traditional Type Inferencing

- Dynamically Typed Languages :

  The effect of constraints may be localised.

  $\Rightarrow$ A little more complicated auxiliary store

- Static type inferencing for dynamically typed languages can be considered a modified version of Hindley-Milner Approach.

Program Property Inference

If constraints are associated with program nodes

- Store bounded by program size

  (As against unbounded auxiliary store)

- Ordering due to *Succ-Pred* relationships

  $\Rightarrow$ Multiple Passes

  - Forward Analysis
    Update attributes at a node to incorporate the effect of constraints in predecessors
    $\Rightarrow$ Attributes of successors may be invalidated
    $\Rightarrow$ Repeat the process for successors
  - Backward Analysis
    Update attributes of predecessor nodes after updating attributes at a node
  - Bidirectional Analysis
    Attributes of both successors as well as predecessors may have to be updated

Space, Time, and Number of Passes

- Description of an algorithm as
  - Single-pass/Multi-pass
  - Forward/Backward/Bidirectional

  is with respect to the allowable storage use

- A Two-pass algo. for Live Variables Analysis

  (*Forward* rather than backward passes!)

  - Identify/store liveness constraints in an auxiliary store (Pass 1)
  - Solve the constraints
  - Write the solutions back (Pass 2)

- No data flow analysis would require more than two passes!

What is Data Flow Analysis?

- A special case of program property inference

  with *bounded store*

- Boundedness of store makes it more practical as compared to general program property inference mechanisms.

- Data flow analysis provides an efficient implicit ordering of solving contraints which would have to be derived explicitly for inference mechanisms.

- Hence data flow analysis continues to thrive as a separate subject of study.

  Thank god! What would have I done otherwise? :-)

A Data Flow Framework for Type Inferencing

A Data Flow Analysis Framework

- Assume type constraints are introduced by
  - Function application node : $x = h(y)$
  - Use node : *Use* x *as* τ

  only.

- Lattice Values : Set of possible types

- Flow functions
  - Function application node : $x = h(y)$
    - Types of h and y constrain the type of x
    - Types of x and y constrain the type of h
    - Types of h and x constrain the type of y
  - Use node : *Use* x *as* τ
    - Type of x is τ
  - All other flow functions are identity functions.

Type Inferencing for Statically Typed Languages

For statically typed languages

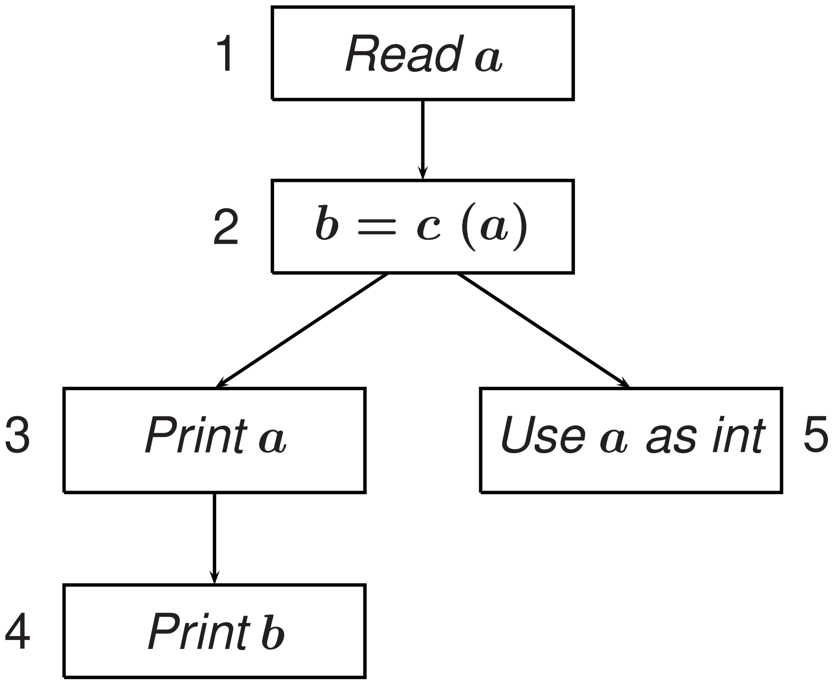
Every variable reference constrains types for all other uses of the variable.

- Initilization : Assume all types are possible
- Analysis : Eliminates types which are not possible
- Resulting Solution for a variable
  - Singleton Set : Unique type
  - Empty Set : (Possibly) inconsistent types
  - Any other Set : Multiple types are possible

  Same solution at all points.

An Example of Type Inferencing for Statically Typed Languages

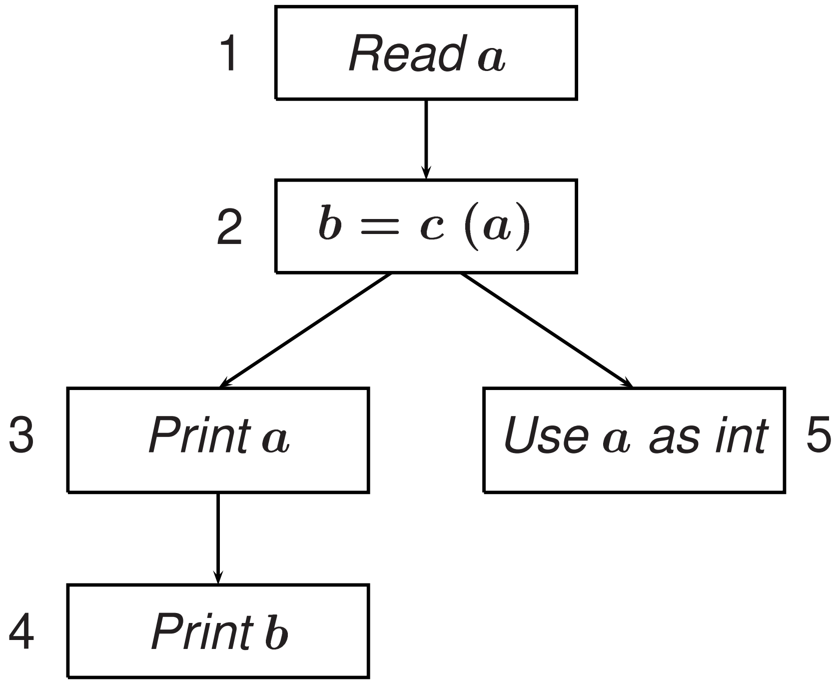
- Types = { int, real, int→int, real→real }
- Reverse Post Order Traversal



- After first traversal

Node	Possible Types		
	a	b	c
2	{i, r}	{i, r}	{i→i, r→r}
4	{i}	{i}	{i→i}

Type Inferencing is Bidirectional



- Types will be constrained at other nodes in subsequent (multiple) traversals.
- Without bidirectional dependencies,

  $\Rightarrow$ Type of a at node 5 cannot be used to constrain types at other nodes.

  $\Rightarrow$ $\boxed{\text{Bidirectionality adds power.}}$
- A workaround can be employed for statically typed languages but not for static type inferencing of dynamically typed languages.

Static Type Inferencing for Dynamically Typed Languages

Why static type inferencing for dynamically typed languages?

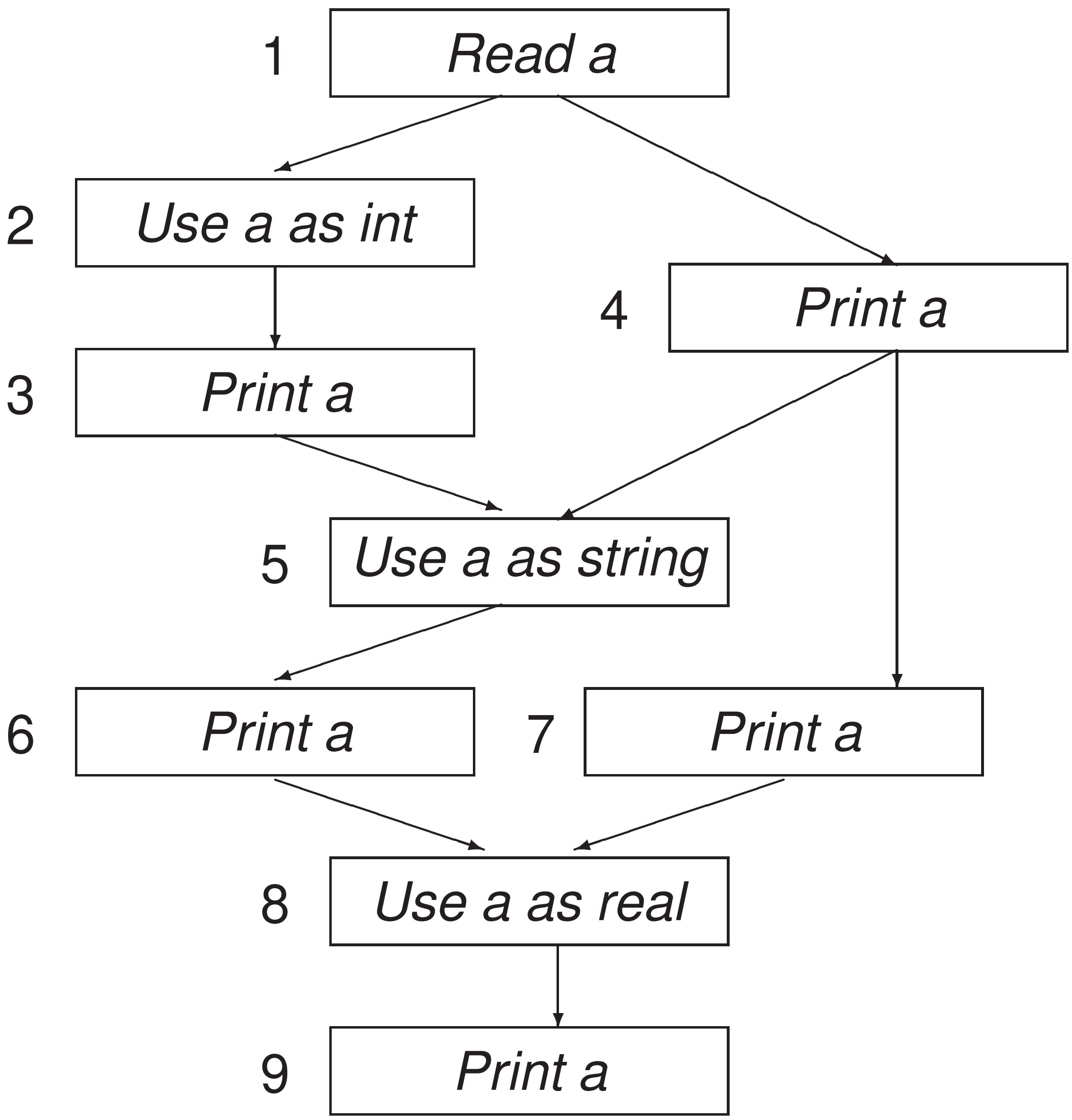
For dynamically typed languages

Usage of varibles enables types for variables.

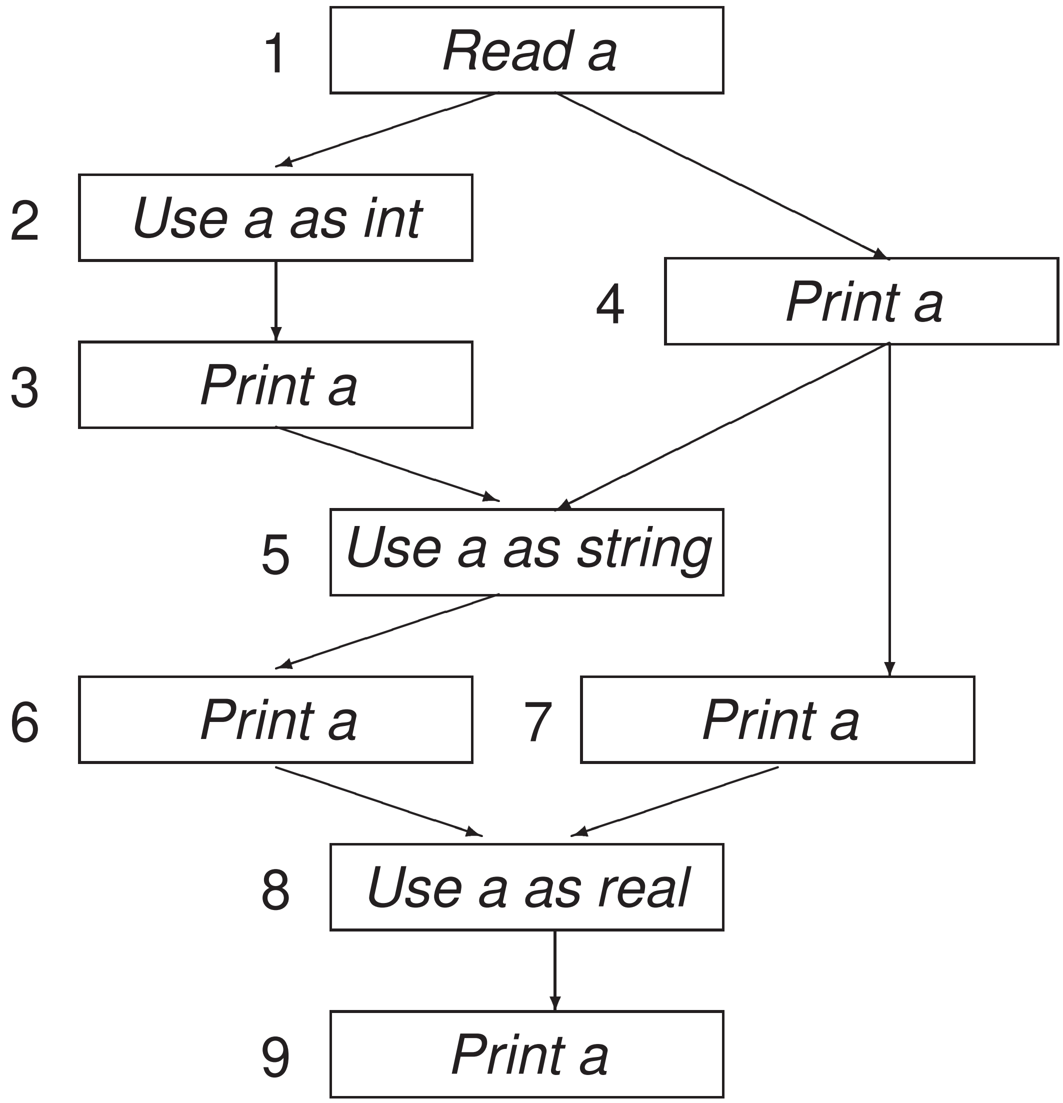
- Initilization : Assume no type is possible
- Analysis : Add types which are possible
- Resulting Solution for a variable
  - Singleton Set : Unique type
  - Empty Set : No type could be inferred
  - Any other Set : Multiple types are possible

  Potentially different solution at different points.
- The need of an error type

Precise Dynamic Type Inferencing



Is This Precise Dynamic Type Inferencing?



- Conclusion : Set of possible types of a is $\{$int,real,string$\}$ everywhere.

Defining A More Precise Confluence Operation

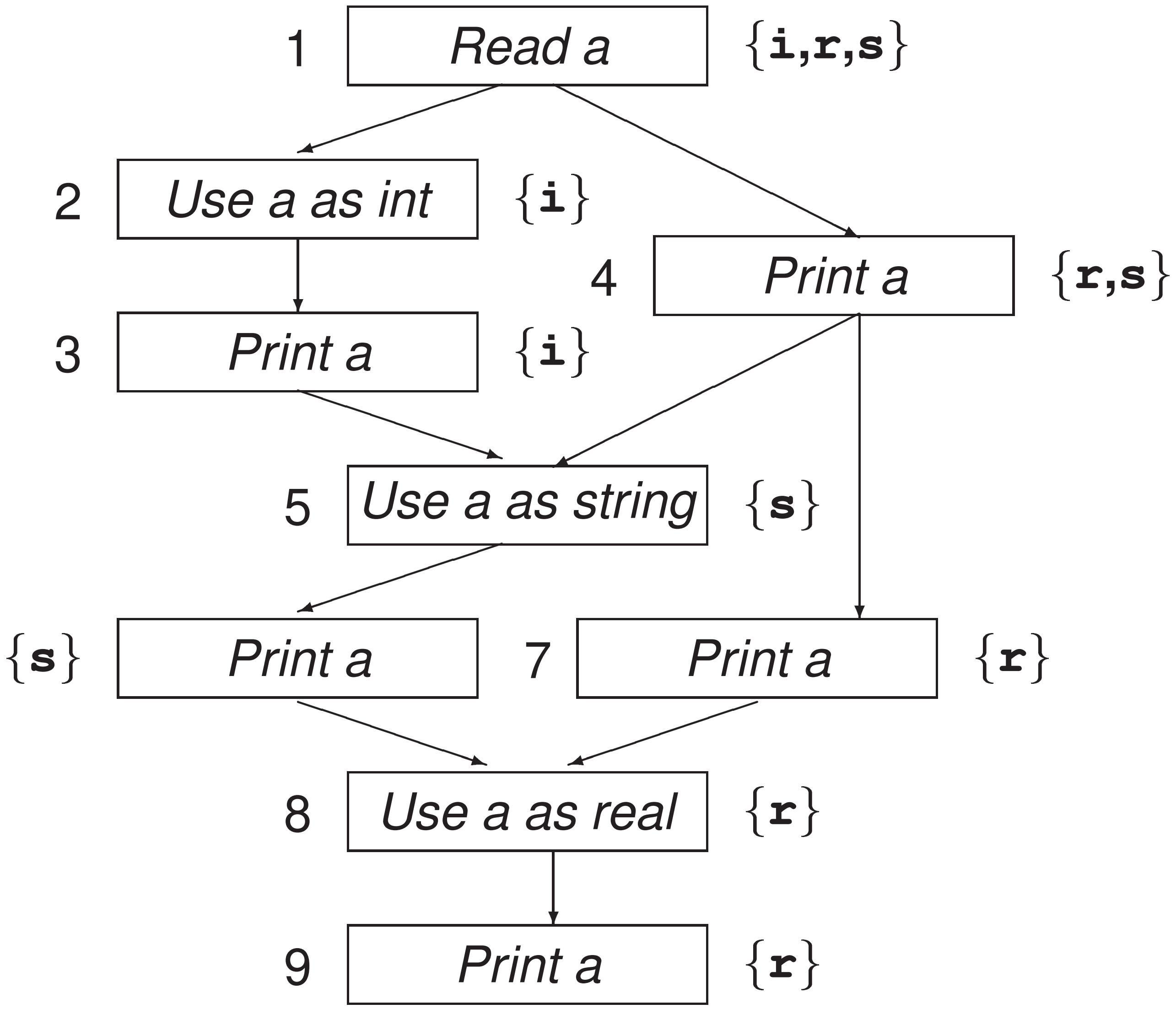
- *Indiscriminate* union $\Rightarrow$ imprecision.
- Solution : *Customise* the union operation.
- Selection Criteria :

  Type τ is included in the set of possible types of x at node n

  If x is referenced as τ in n
  else if x is referenced as τ in an ancestor of n
  else if x is referenced as τ in a descendant of n
  else if x is referenced as τ in any other node.

  $\Rightarrow$ Multi-level forward and backward dependencies

Precise Dynamic Type Inferencing



Precise Dynamic Type Inferencing

- $\uplus$ so defined is

  Commutative, Associative, and Idempotent.

- The resulting partial order is

  Transitive, Antisymmetric, and Reflexive.

- The lattice has a $\top$ and a $\perp$ element.

  (Assumption : The set of types is fixed)

Applications to Dateness Analysis

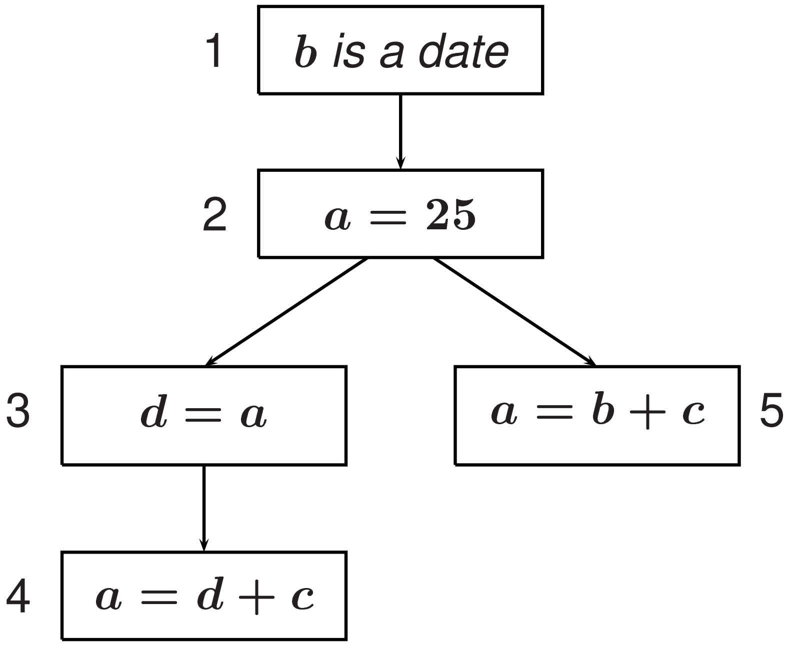
Special case of Dynamic Typing and *Unit of Measure Analyses*

- Types = $\{$ date, int, error $\}$
- Flow Functions for $x = y + z$

$$\text{Type of } x = \begin{cases} \text{error} & \text{if type of } y \text{ or } z \text{ is error} \\ \text{date} & \text{if ((} y \text{ is date and } z \text{ could be int)} \\ & \quad \text{OR} \\ & \quad (z \text{ is date and } y \text{ could be int))} \\ \text{int} & \text{if } y \text{ and } z \text{ both are int} \\ \text{unknown} & \text{otherwise} \end{cases}$$

$$\text{Type of } y = \begin{cases} \text{error} & \text{if type of } x \text{ or } z \text{ is error} \\ \text{date} & \text{if ((} x \text{ is date and } z \text{ could be int)} \\ \text{int} & \text{if ((} x \text{ and } z \text{ are int)} \\ & \quad \text{OR} \\ & \quad (x \text{ and } z \text{ are dates))} \\ \text{unknown} & \text{otherwise} \end{cases}$$

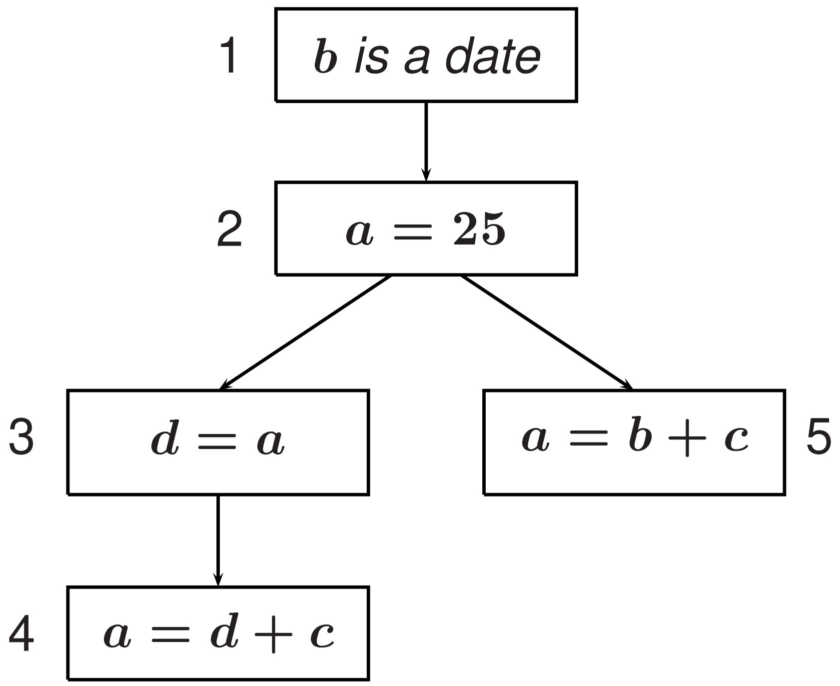
An Example of Dateness Analysis



- After first traversal

Node	Possible Types			
	a	b	c	d
1	$\emptyset$	{ d}	$\emptyset$	$\emptyset$
2	$\emptyset$	{ d}	$\emptyset$	$\emptyset$
3	$\emptyset$	{ d}	$\emptyset$	$\emptyset$
4	$\emptyset$	{ d}	$\emptyset$	$\emptyset$
5	{ d }	{ d}	{ d}	$\emptyset$

Dateness Analysis



- Resulting Solution : a, b, c are dates. d is int.
- Can be derived after propagating information from node 5 to 4.

  $\Rightarrow$ Bidirectional dependencies.
- Unidirectional formulation cannot discover that d is int and cannot be a date in node 4.

  $\Rightarrow$ $\boxed{\text{Bidirectionality adds power.}}$

Conclusions

- $\boxed{\text{Bidirectional data flow analyses } \textit{do} \text{ exist.}}$
  - If unbounded storage is allowed
    $\Rightarrow$ No data flow analysis would need more than two passes!
  - If unbounded storage is disallowed
    $\Rightarrow$ Type inferencing is bidirectional
    (We didn't see it that way.)
- A formal theory for bidirectional data flow analysis exists.

  (Some extensions are required.)
- Bidirectionality $\rightarrow$ Powerful Mechanism
  - Don't avoid it.
  - Welcome it and Overcome it.
    (and read my papers :-))

Acknowledgements

- Other Contributors :

  D. M. Dhamdhere and Alan Mycroft

  Dateness application suggested by A. Sanyal

- Talk based on
  - *Bidirectional Data Flow Analysis : Myths and Reality*
    Khedker & Dhamdhere, SIGPLAN Notices (June 99).
  - *Bidirectional Data Flow Analyses Do Exist*
    Khedker, Dhamdhere, and Mycroft (Submitted)