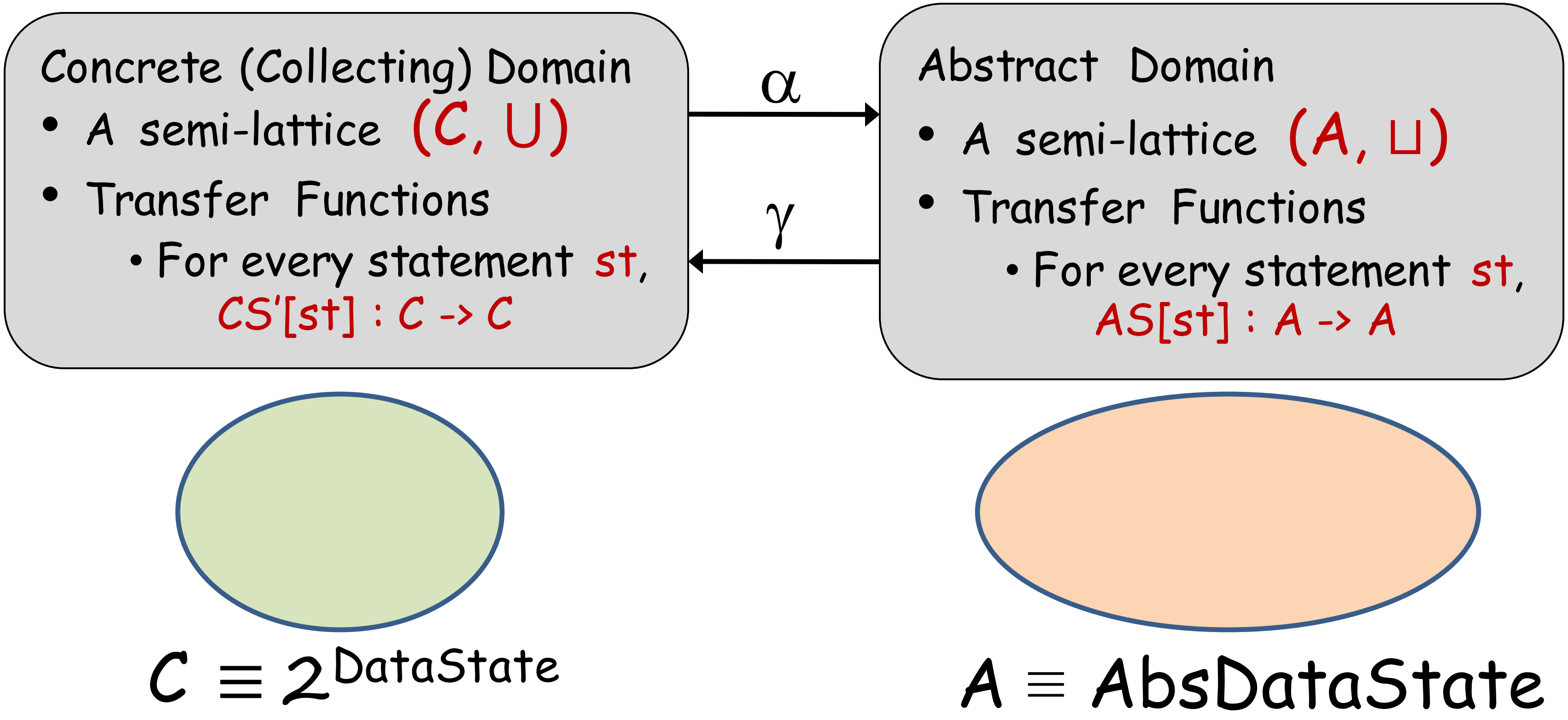
Pointer Analysis

Lecture 2

G. Ramalingam
Microsoft Research, India
&
K. V. Raghavan

Correctness and precision of Algorithm A

Enter: The French Recipe (Abstract Interpretation)



Concrete (Collecting) Domain

- A semi-lattice $(C, \cup)$
- Transfer Functions
  - For every statement st, CS'[st] : C -> C

Abstract Domain

- A semi-lattice $(A, \sqcup)$
- Transfer Functions
  - For every statement st, AS[st] : A -> A

$C \equiv 2^{DataState}$

$A \equiv AbsDataState$

Points-To Analysis (Abstract Interpretation)

$\alpha(Y)$ = \p. {x | exists s in Y. s(p) == x }

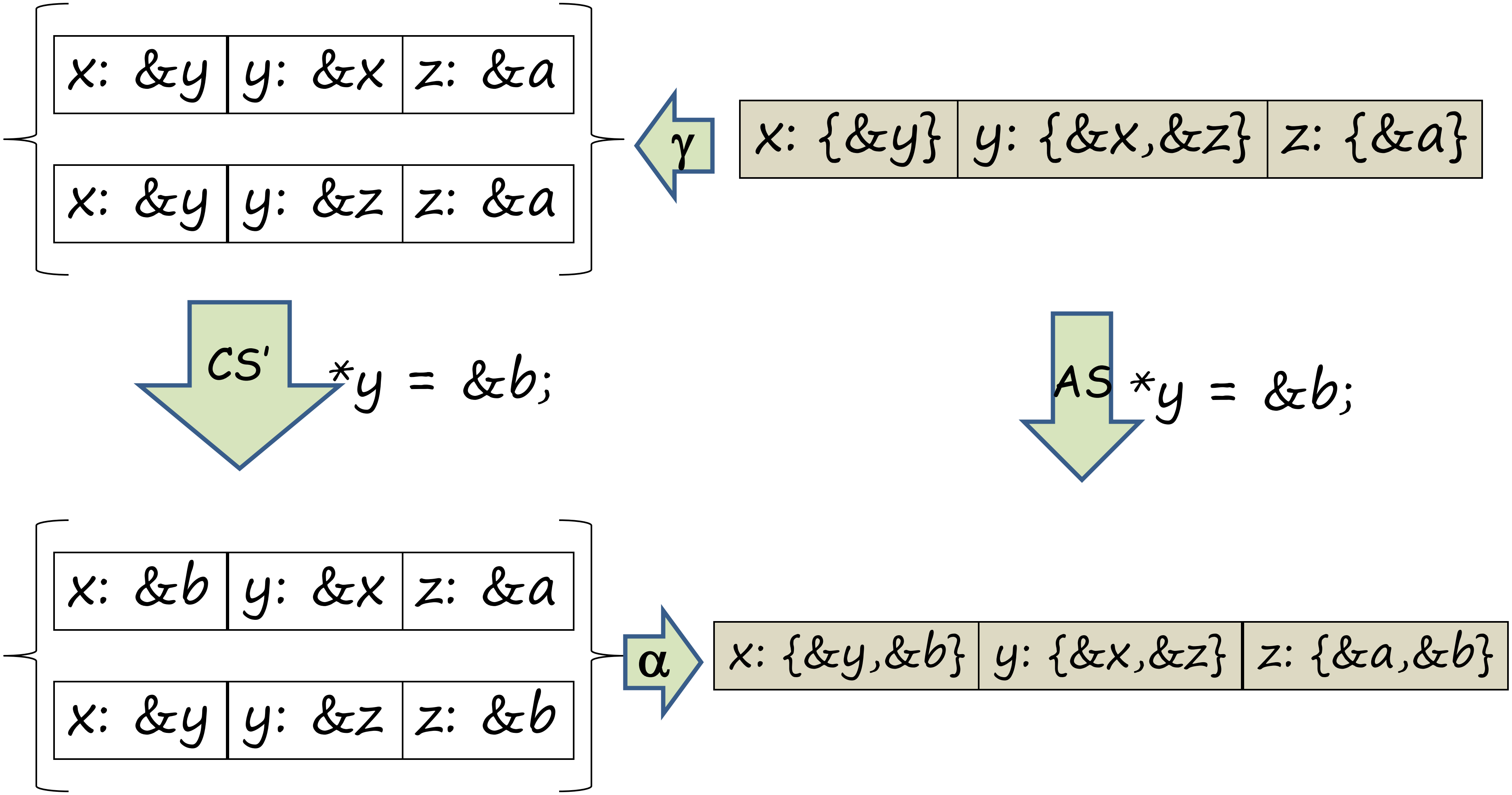
$\gamma(a)$ = {s | for each pointer variable p, s(p) $\in$ a(p)}

Approximating Transformers: Correctness Criterion

It can be shown that for any statement st, and for any $a1 \in A$

$$AS[st](a1) \geq \alpha\,(CS[st](\,\gamma(a1))$$

Correctness illustration



x: &y	y: &x	z: &a

x: &y	y: &z	z: &a

γ

x: {&y}	y: {&x,&z}	z: {&a}

CS' *y = &b;

AS *y = &b;

x: &b	y: &x	z: &a

x: &y	y: &z	z: &b

α

x: {&y,&b}	y: {&x,&z}	z: {&a,&b}

Is The Precise Solution Computable?

- Claim: The set RS(u) of reachable concrete states (for our language) is precisely computable.
  - (However, Algorithm A is imprecise)

- Note: This is true for any collecting semantics with a finite state space.
- This is true only for restricted language!

Precise Points-To Analysis: Computational Complexity

- What's the complexity of the least-fixed point computation using the collecting semantics?

- The worst-case complexity of computing reachable states is exponential in the number of variables.
  - Can we do better?

- Theorem: Computing precise may-point-to is PSPACE-hard even if we have only two-level pointers.

Precise Points-To Analysis: Caveats

- Theorem: Precise may-alias analysis is undecidable in the presence of dynamic memory allocation.
  - Add "x = new/malloc ()" to language
  - State-space becomes infinite

- Digression: Integer variables + conditional-branching involving integer variables also makes any precise analysis undecidable.

Dynamic Memory Allocation

- s: x = new () / malloc ()
- Assume, for now, that allocated object stores one pointer
  - s: x = malloc (sizeof(void*))

- Introduce a pseudo-variable V_s to represent objects allocated at statement s, and use previous algorithm
  - treat s as if it were "$x = \&V_s$"
  - also track possible values of V_s
  - allocation-site based approach

α in the presence of pseudo variables

$\alpha(Y)$ = \p. {x | exist s in Y such that s(y) = z
such that:
((p is a normal variable and y is p) OR
(p is V_r and y is an address
allocated at site r))
AND
((x is a normal variable and x is z) OR
(x is V_t and z is an address
allocated at site t)) }

(For simplicity, assume that the set of all concrete addresses is partitioned upfront among all allocation sites in the program)

γ in the presence of pseudo variables

$\gamma(a) = \{s \mid \forall$ normal variables p:

$s(p) = x \land x$ is a normal variable $\land\ x \in a(p)$, OR

$s(p) = y \land y$ is an address allocated at

site $t \land V_t \in a(p)$, AND

$\forall$ pseudo-variables V_r: $\forall$ addresses y allocated at V_r:

$s(y) = x \land x$ is a normal variable $\land\ x \in a(V_r)$, OR

$s(y) = z \land z$ is an address allocated at

site $t \land V_t \in a(V_r)\}$

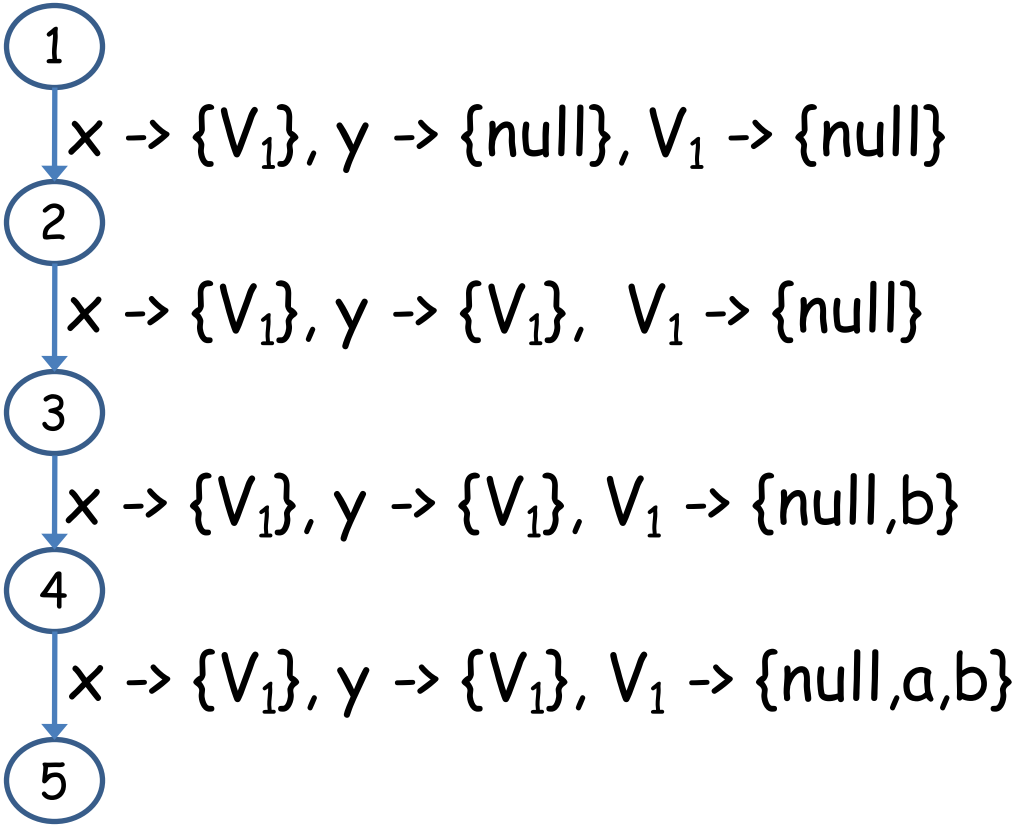
Dynamic Memory Allocation: A run of the algorithm

```
x = new; // 1

y = x;

*y = &b;

*y = &a;
```



x -> $\{V_1\}$, y -> $\{null\}$, V_1 -> $\{null\}$

x -> $\{V_1\}$, y -> $\{V_1\}$, V_1 -> $\{null\}$

x -> $\{V_1\}$, y -> $\{V_1\}$, V_1 -> $\{null,b\}$

x -> $\{V_1\}$, y -> $\{V_1\}$, V_1 -> $\{null,a,b\}$

Illustrating need for weak updates on pseudo variables

- Key aspect: V_s represents a set of memory locations, not a single location
  - if x->{V_s}, to be safe "*x = .." still needs weak update!
- Consider this program:

```
do {x = new /* V1 */; *x = &a} while(..);
*x = &b;
```

Exercise: Say in the last stmt above we set V_1 -> {b}. Show that \gamma(V_1 -> {b}) does not include all concrete states that can arise at the end of the program.

Inter-procedural analysis

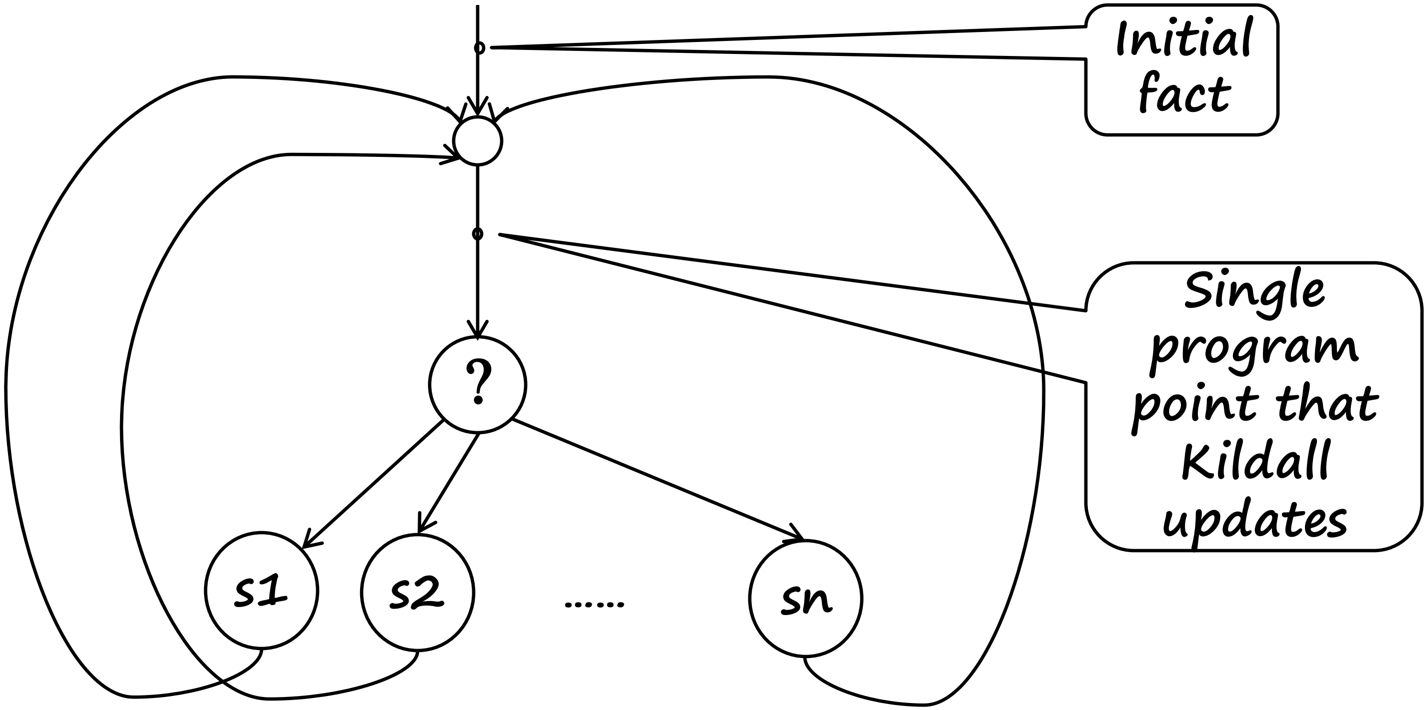
- Context-sensitivity can be achieved using standard techniques
- Indirect (virtual) function call sites need to be resolved to candidate functions using points-to analysis. And points-to analysis needs calls to be resolved! Therefore, the two have to happen hand in hand.

Andersen's Analysis

- A *flow-insensitive* analysis
  - computes a single points-to solution, which over-approximates points-to solutions at all program points
  - ignores control-flow – treats program as a set of statements
  - equivalent to collapsing the given program to have a single program point, and then applying Algorithm A on it.

Andersen's Analysis

If program has statements s1, s2, …, sn, then create collapsed CFG as follows:



Initial fact

Single program point that Kildall updates

After algorithm terminates, final solution at the single program point over-approximates result computed by flow-sensitive analysis at any point

Example:
Andersen's Analysis

```
x = &a;

*x = &w;

y = x;

x = &b;

*x = &t;

z = x;
```

Before first iteration: all variables null

After first iteration of Kildall:

X -> {a,b,null}, all other variables null

After 2nd iteration:
X -> {a,b,null}, y,z -> {a,b,null}, a,b -> {w,t,null}, all other variables null

After 3rd iteration:
X -> {a,b,null}, y,z -> {a,b,null}, a,b -> {w,t,null}, all other variables null

Notes about Andersen's Analysis

- Strong updates never happen in Andersen's analysis!
  - If x->{y} and y->{w} before we process statement "*x = &z", then even if transfer function returns y->{z}, due to subsequent join, y will point to {w,z} after this step.
- Flow-insensitive style can be adopted for any analysis, not just for pointer analysis

Why Flow-Insensitive Analysis?

- Reduced space requirements
  - a single points-to solution
- Reduced time complexity
  - no copying of points-to facts
    - individual updates more efficient
  - a cubic-time algorithm
- Scales to millions of lines of code
  - most popular points-to analysis

Andersen's Analysis: An alternative formulation

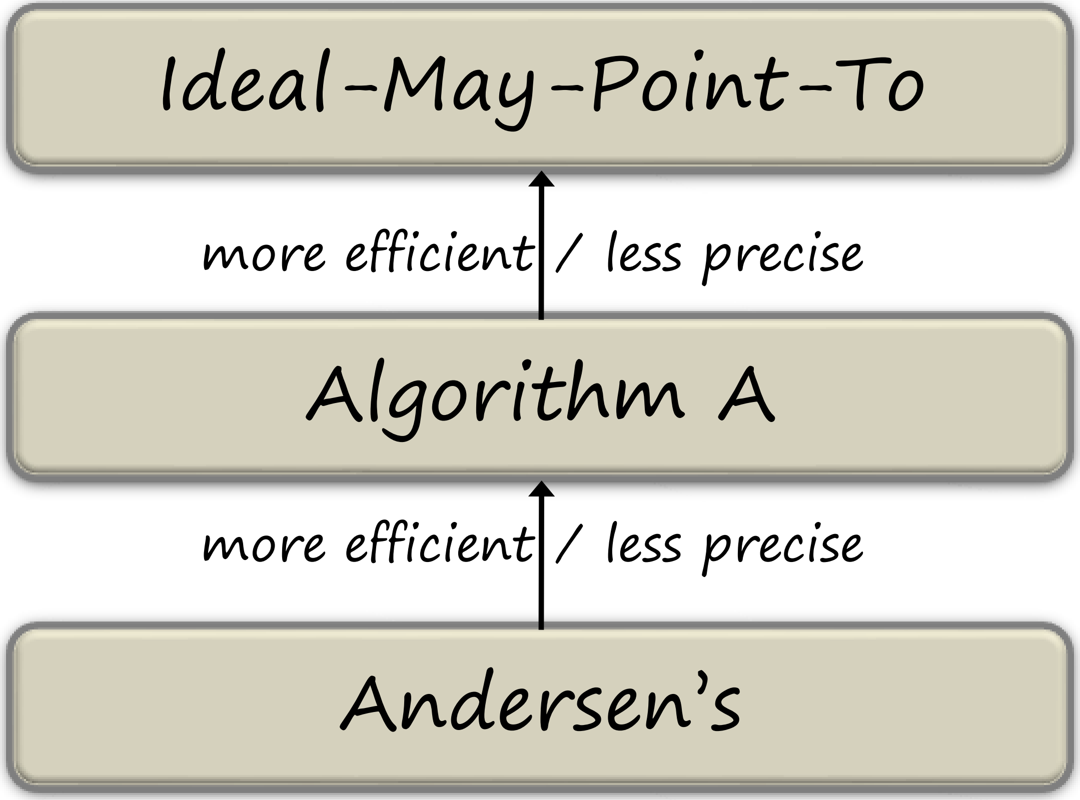
1. Introduce a constraint variable PT_x for each program variable x
2. Create a constraint from each assignment statement, as follows:

   x = y: $PT_x \subseteq PT_y$
   *x = y: $PT_v \subseteq PT_y$, forall variables v in PT_x
   x = &y: $PT_x \subseteq \{y\}$
   x = *y: $PT_x \subseteq PT_v$, forall variables v in PT_y

3. Find least solution to set of all constraints that were generated above. (A solution is a mapping from constraint variables to sets of program variables.) Emit this least solution as the final solution.
   - Note: Solution s1 dominates Solution s2 if for each program variable v, s2(PT_v) $\subseteq$ s1(PT_v)

Note: This approach computes exact same result as previous approach that collapses program and then uses Algorithm A.

May-Point-To Analyses



Ideal-May-Point-To

more efficient / less precise

Algorithm A

more efficient / less precise

Andersen's

Andersen's Analysis: Further Optimizations and Extensions

- Fahndrich et al., Partial online cycle elimination in inclusion constraint graphs, PLDI 1998.
- Rountev and Chandra, Offline variable substitution for scaling points-to analysis, 2000.
- Heintze and Tardieu, Ultra-fast aliasing analysis using CLA: a million lines of C code in a second, PLDI 2001.
- M. Hind, Pointer analysis: Haven't we solved this problem yet?, PASTE 2001.
- Hardekopf and Lin, The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code, PLDI 2007.
- Hardekopf and Lin, Exploiting pointer and location equivalence to optimize pointer analysis, SAS 2007.
- Hardekopf and Lin, Semi-sparse flow-sensitive pointer analysis, POPL 2009.

Context-Sensitivity Etc.

- Liang & Harrold, Efficient computation of parameterized pointer information for interprocedural analyses. SAS 2001.
- Lattner et al., Making context-sensitive points-to analysis with heap cloning practical for the real world, PLDI 2007.
- Zhu & Calman, Symbolic pointer analysis revisited. PLDI 2004.
- Whaley & Lam, Cloning-based context-sensitive pointer alias analysis using BDD, PLDI 2004.
- Rountev et al. Points-to analysis for Java using annotated constraints. OOPSLA 2001.
- Milanova et al. Parameterized object sensitivity for points-to and side-effect analyses for Java. ISSTA 2002.

Applications

- Compiler optimizations
- Verification & Bug Finding
  - use in preliminary phases
  - use in verification itself