Iterative Program Analysis Abstract Interpretation

Summary by Ben Riva & Ofri Ziv

Soundness Theorem

Theorem: If a computation fixed-point is sound, then its least-fixed-point is sound.
More precisely, **IF**:

- (α, γ) forms a Galois connection from a concrete domain C to an abstract domain A.
- $f: C \to C$ is a monotone function describing the small step semantics of the program (possibly uncomputable).
- $f^{\#}: A \to A$ is a monotone function in the abstract domain (computable).
- Any of the following:
  - $\forall a \in A: f(\gamma(a)) \sqsubseteq \gamma(f^{\#}(a))$
    Concrete computation is more precise than the concretization of an abstract computation (abstract computation might lose information).
  - $\forall c \in C: \alpha(f(c)) \sqsubseteq f^{\#}(\alpha(c))$
    Abstraction of a concrete step is more precise than its equivalent step abstraction.
  - $\forall a \in A: \alpha(f(\gamma(a))) \sqsubseteq f^{\#}(a)$
    $f^{\#}$ is the best transformer (the abstraction precision's lower bound).

THEN:

- $lfp(f) \sqsubseteq \gamma(lfp(f^{\#}))$- The concrete domain's least-fixed-point (lfp) is more precise than the abstract domain's concretized lfp (it contains less options).
- $\alpha(lfp(f)) \sqsubseteq lfp(f^{\#})$ - The abstraction of the concrete domain's lfp is more precise than the abstract domain's lfp.

According to Cousot theorem, the abstract domain's solution gives us an approximation of the concrete domain's solution, but it might lose some information.

Completeness

An analysis completeness is measured by two (non-equivalent) equations:

1. Abstraction completeness - $\alpha(lfp(f)) = lfp(f^{\#})$
   Abstract computation is equivalent to the computation in the abstraction of the concrete domain (abstraction does not lose information).
2. Concretization completeness -$lfp(f) = \gamma(lfp(f^{\#}))$
   Computation in the concretization of the abstract domain is equivalent to the computation in the concrete domain. Meaning the abstraction describes only the concrete domain's reachable values.

EXAMPLES

1) Constant propagation (note that there is no information loss, and the analysis will return the precise answer):

```
x := 1
while()
        skip
```

2) Correlated ifs:

```
if (<>):
        x := 3
else:
        x := 2
/* POINT 1 */
if (x == 2):
        x := 3
/* POINT 2 */
```

Lets analyze the example at different points:

- **/* POINT 2 */**
  Concrete solution - *x=3*
  Abstract solution - *x=?*
  Both of the completeness equations aren't valid.
- **/* POINT 1 */**
  Concrete solution - *x=?*
  Abstract solution - *x=?*
  The abstraction is complete (since lfp of {2,3} is ?).
  On the other hand, the concretization isn't complete.

Application of AI: Constant Propagation

In constant propagation we search for variables with constant values at some point of the program. This can be done using abstract interpretation with the following functions:

- $\beta: [Var \to Z] \to [Var \to Z \cup \{\top, \bot\}]$
  - $\beta(\sigma) = \sigma$

  β maps a state (mapping of variables to integer values) to an abstract state (mapping of variables to integer values and $\{\top, \bot\}$).
  In the constant propagation case it is the identity function.
- $\alpha: P([Var \to Z]) \to [Var \to Z \cup \{\top, \bot\}]$
  - $\alpha(X) = \sqcup \{\beta(\sigma) | \sigma \in X\} = \sqcup \{\sigma | \sigma \in X\}$

  α maps a group of concrete states to a single abstract state. This mapping is done using the join operator and in the constant propagation case it will follow those rules:
  - A variable that on different states is assigned with different values, is mapped to $\top$.

    - A variable that was never assigned is mapped to $\bot$.
    - A variable is mapped to a constant value, if this value or the $\bot$ was assigned to it on all states.
- $\gamma: [Var \to Z \cup \{\top, \bot\}] \to P([Var \to Z])$
    - $\gamma(\sigma^{\#}) = \{\sigma | \beta(\sigma) \sqsubseteq \sigma^{\#}\} = \{\sigma | \sigma \sqsubseteq \sigma^{\#}\}$

  γ is the concretization function. It maps an abstract state to a group of states, which contains all the concrete states whose abstraction will lead back to the abstract state, i.e., $\gamma(\bot) = \phi, \gamma(\top) = all\ elements$

Local soundness

A computation is called *locally-sound* if there is no unknown behaviour. This is obtained by implying all statements on all possible values, or more formally: $[[st]]^{\#}(\sigma^{\#}) \sqsupseteq \alpha(\{[[st]]\sigma | \sigma \in \gamma(\sigma^{\#})\}) = \sqcup \{[[st]]\sigma | \sigma \sqsubseteq \sigma^{\#}\}$

Meaning that abstraction of a concrete computation is more precise than an abstract computation over an abstract state.

Optimality

A computation is *optimal* if an abstraction of a concrete computation is exactly as precise as an abstract computation over an abstract state:
$[[st]]^{\#}(\sigma^{\#}) = \alpha(\{[[st]]\sigma | \sigma \in \gamma(\sigma^{\#})\}) = \sqcup \{[[st]]\sigma | \sigma \sqsubseteq \sigma^{\#}\}$

Constant propagation as defined above is not optimal: For instance, expressions like x*0 should yield 0, while in our analysis if x is $\top$, it returns $\top$.

Soundness

According to Cousot's theorem, local soundness implies global soundness, therefore **constant propagation defined above is sound**.

Completeness

Since abstract computation of constant propagation, as defined above, does not yield the same results as the abstraction of the concrete computation, it is **not complete**.

Example: Interval Analysis

Objective: Find a variable's lower and upper bounds. For instance, this analysis is used to verify arrays indexes are not out of bound.

We define the following lattice:
$L = (Z \cup \{-\infty, \infty\} \times Z \cup \{-\infty, \infty\}, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$

- $[a,b] \sqsubseteq [c,d]\ if\ c \leq a\ and\ d \geq b$
- $[a,b] \sqcup [c,d] = [min(a,c), max(b,d)]$
- $[a,b] \sqcap [c,d] = [max(a,c), min(b,d)]$
- $\bot = \phi$
- $\top = [-\infty, \infty]$

Two join/meet examples:

- $[1,3] \sqcup [10,12] = [min(1,10), max(3,12)] = [1,12]$
  We can notice that the joint operation causes information loss, since it includes many values in the range.
- $[2,5] \sqcap [9,10] = [9,5] = \bot$
  The last equality is derived from the fact that $[9,5] = \phi$.

Now let's define the galois connection for this lattice:

- $\beta: [Var \to Z] \to [Var \to L]$
  $\beta(a) = [a,a]$
- $\alpha: [Var \to Z^2] \to [Var \to L]$
  $\alpha(X) = \sqcup \{\beta(a) | a \in X\} = [minX, maxX]$
- $\gamma: [Var \to L] \to [Var \to Z]$
  $\gamma(Y) = \{a | \beta(a) \sqsubseteq Y\}$ - all sub-ranges
  $\gamma([l,u]) = \{a | l \leq a \leq u\}$

For example, for the *skip* function:

- $[skip]^{\#}([l,u]) = [l,u]$
- *skip* is best transformer:

$skip^{\#}([l,u]) = \alpha([\overline{\text{skip}}](\gamma([l,u]))\ where\ \overline{\text{skip}}(X) = \{skip(a) | a \in X\}$
($\overline{skip}$ is the activation of the skip function on every element in the group.)

Domains with Infinite Heights for Integers

The interval example is useful in simple scenarios. However, when variables have more complex relations, even between themselves, other techniques are needed. Few known techniques are:

- Intervals such as $\pm x_i \leq b$ [Static Determination of Dynamic Properties of Programs. Patrick Cousot and Radhia Cousot, 1976]
- Octagons $\pm x_i y_i \leq b$ [(Automatic removal of array memory leaks in java. Ran Shaham, Elliot K. Kolodner and Shmuel Sagiv, 2000), (The Octagon Abstract Domain. Antoine Mine, 2001)]
- Polyhedra $\sum a_i * x_i \leq b$ [Automatic discovery of linear restraints among variables of a program. Patrick Cousot and Nicolas Halbwachs, 1978]

Application of AI: Formal Available Expressions

Let's see another application of abstract interpretation. In many cases, the compiler produces expressions that we would like to reuse on different points in the program in order to reduce calculations. More specifically, we would like to find out what expressions are available at a given program point.

For example:

```
x = y + t                 //{(y+t)}
z = y + r                 //{(y+t),(y+r)}
while (...) {             //{(y+t),(y+r)}
        t = t + (y + r)   //{(y+r)}
}
```

In the comment of each line we wrote the available expressions at that point. In the last line, the only expression left was *y+r*, (without *y+t* and *t+(y+r)*) since variable *t* is changed on this line, leaving all the "t dependent expressions" unusable. (Note that it includes t+(y+r) since that expression needs a reevaluation).

Available expression lattice

A formal definition of the problem would be:

- P(Fexp)
- $X \sqsubseteq Y \Leftrightarrow X \supseteq Y$. Note that X is more precise than Y means that X contains all of Y's expressions.
- $X \sqcup Y = X \cap Y$
- $\perp = Fexp$
- $\top = \phi$

The available expression problem is a "must problem", meaning that we are interested in expressions that are available from all courses. The domain of this problem is the power set of all the program's expressions Fexp.

While available expressions

A computation of Fexp in WHILE program is performed according to the following definitions:

- $s::= skip$
- $s::= id := exp \; \{s.Fexp = \{exp.rep\}\}$
- $s::= s_1; s_2 \; \{s.Fexp := s_1.Fexp \cup s_2.Fexp\}$
- $s::= while \; exp \; do \; s_1 \; \{s.Fexp := \{exp.rep\} \cup s_1.Fexp\}$
- $s::= if \; exp \; then \; s_1 \; else \; s_2 \; \{s.Fexp := \{exp.rep\} \cup s_1.Fexp \cup s_2.Fexp\}$

Instrumented Semantics Available expressions

Instrumented semantics analyze all the states and Fexps (all program's expressions) in order to produce the available expressions set at each line of the program. The semantics' formal definition is as follows:

- $S[[Stm]]: State \times P(Fexp) \rightarrow State \times P(Fexp)$
  $State = (\sigma, ae)$, where σ is the "variables states" and ae is the "available expressions".
- $S[[id := a]](\sigma, ae) = (\sigma[a \rightarrow A[[a]]\sigma], notArg(id, ae \cup \{a\})$
  (*notArg* eliminates all the expressions containing the variable *id*, which is the variable changed by this statement.)
  Assignment leads to the following state:
  - "Variables states" as before, only with the *id* variable updated value.
  - "Available expressions" consists of *ae* (the "old" available expressions set) and *a* (the expression appeared in the assignment statement), minus all the expressions that contain the *id* variable. This last operation is done since the assignment statement changed *id*'s value, and by that made all those expressions to be invalid (i.e., need to be recomputed).
- $S[[skip]](\sigma, ae) = (\sigma, ae)$
  skip statement leaves the state as is.

Collecting Semantics

We defined semantics on each of the language's non-control-flow statements. Now we define the collecting semantics, which activates all the program statements over all available states.

- $CS[[Stm]]: P(State \times P(Fexp)) \rightarrow P(State \times P(Fexp))$
- $CS[[s]](X) = \{S[[s]](\sigma, ae) | (\sigma, ae) \in X\}$

Example

Let's use the collecting semantics on the following example, with the initial state $([x \rightarrow 0, y \rightarrow 1, z \rightarrow 6, r \rightarrow 5, t \rightarrow 5], \phi)$:

x = y * z;	$([x \rightarrow 6, y \rightarrow 1, z \rightarrow 6, r \rightarrow 5, t \rightarrow 5], (y * z))$
if (x == 6)	
y = z + t;	$([x \rightarrow 6, y \rightarrow 11, z \rightarrow 6, r \rightarrow 5, t \rightarrow 5], (z + t))$
…	
if (x == 6)	
r = z + t;	$([x \rightarrow 6, y \rightarrow 11, z \rightarrow 6, r \rightarrow 11, t \rightarrow 5], (z + t))$

Formal Available Expressions Abstraction

- $\beta: [Var \rightarrow Z] \times P(Fexp) \rightarrow P(Fexp)$
  $\beta(\sigma, ae) = (ae)$
  The abstract interpretation of a concrete state is the concrete's group of available expressions.
- $\alpha: P(P([Var \rightarrow Z] \times P(Fexp)) \rightarrow P(Fexp)$
  $\alpha(X) = \sqcup \{\beta(\sigma) | \sigma \in X\} = \cap \{ae | (\sigma, a) \in X\}$

The abstract interpretation of a concrete states' group is a group of available expressions all states agree on.

- $\gamma: P(Fexp) \to P(P([Var \to Z] \times P(Fexp))$
  $\gamma(ae^{\#}) = \{(\sigma, ae) | \beta(\sigma, ae) \sqsubseteq ae^{\#}\} = \{(\sigma, ae) | ae \supseteq ae^{\#}\}$
  The concretization of an abstract state is a group of concrete states, which their abstract interpretation contains at least all the given abstract state's available expressions.

Formal Available Expressions Abstract Interpretation

After describing the abstraction, we define the semantics:

- $S[[Stm]]^{\#}: P(Fexp) \to P(Fexp)$
- $S[[id := a]]^{\#}(ae) = notArg(id, ae \cup \{a\})$
  Assignment statement adds the calculated assignment to the available expressions group, and remove all expressions containing the *id* variable, since the statement changed its value and caused all the expressions it appeared in to be invalidate.
- $S[[skip]]^{\#}(ae) = (ae)$
  skip statement doesn't affect the available expressions group.
- All other While language statements determine control-flow, therefore are irrelevant.

Local Soundness

$[[st]]^{\#}(ae^{\#}) \supseteq \sqcup \{[[st]](\sigma, ae)[1] | ae \sqsubseteq ae^{\#}\}$
Note: $[[st]](\sigma, ae)[1] = ae_1 \; where \; [[st]](\sigma, ae) = (\sigma_1, ae_1)$.
The abstract interpretation we defined for the available expressions problem is locally sound, and therefore by Cousot's theorem it is globally sound.

Optimality

The abstract interpretation above is also optimal, or more formally it maintains:
$[[st]]^{\#}(ae^{\#}) = \alpha(\{[[st]](\sigma, ae) | (\sigma, ae) \in \gamma(\sigma^{\#})\}) = \sqcup \{[[st]](\sigma, ae)[1] | ae \sqsubseteq ae^{\#}\}$

Example program

```
x = y + t              //{(y+t)}
z = y + r              //{(y+t),(y+r)}
while (...) {          //{(y+t),(y+r)}
        t = t + (y + r) //{(y+r)}
}
```

Explanation:

- We begin the analysis with no available expressions ($ae = \top = \phi$).
- After analyzing the while statement we merge its result with the group of available expressions before it, and receive: $\{(y + t), (y + r)\} \sqcup \{(y + r)\} = \{(y + r)\}$

Completeness

Let's see an abstract interpretation example on a previous program. We begin the analysis with nothing computed - ϕ.

```
x = y * z;              //[y*z]
if (x == 6)
        y = z + t;      //[z+t]
                        //When we finish with the if block execution, we merge the //result with
                        main using join - [y * z] ⊔ [z + t] = ϕ
…
if (x == 6)
        r = z + t;      //[z+t]
```

We can note that this analysis is much less precise than the collecting semantics analysis done earlier. At each line we have at most the same expressions as in the collecting semantics. The analysis doesn't deduce that in case one of the if-blocks is carried out, then also the other one is (assuming *x* doesn't change in between).
In the abstract interpretation, after the first if-block we had no available expressions, while in the collecting semantics example we showed before there was one available expressions at this point. Since our abstract interpretation missed an available expression it is <u>not</u> complete.

Application of AI: May-Be-Garbage

A variable *x* may-be-garbage at a program point *v* if there exists an execution path leading to *v* in which *x*'s value is unpredictable:

- *x* not assigned
- *x* assigned using an unpredictable expression

As usual, we define the following:

- The lattice is $L = (P(Var), \sqsubseteq{=}\subseteq, \sqcup{=}\cup, \sqcap{=}\cap, \bot{=}\phi, \top = Var)$
- The abstract interpretation is:
  - $\beta: [Var \to Z] \times P(Var) \to P(Var)$

  $\beta(\sigma, a) = a$
  - $\alpha$and γ are defined accordingly.
- The semantics is (we use C syntax for short if-else)

  $[[x := a]]^{\#}(g) = [(arg(a) \cap g \neq \phi)\ ?\ (g \cup \{x\})\ |\ (g - \{x\})]$

  where *g* is all the "garbage" variables, and *arg(a)* is the group of variables in the expression *a*.

  This function is monotone since if $g_1 \sqsubseteq g_2$, then $[[x := a]]^{\#}(g_1) \sqsubseteq [[x := a]]^{\#}(g_2)$.

We now prove soundness. By definition,
$[[x := a]](\sigma, g) = (\sigma[x \to A[[a]]\sigma], [(arg(a) \cap g \neq \phi)\ ?\ (g \cup \{x\})\ |\ (g - \{x\})])$
On the other hand, $[[x := a]]^{\#}(g^{\#}) = [(arg(a) \cap g^{\#} \neq \phi)\ ?\ (g^{\#} \cup \{x\})\ |\ (g^{\#} - \{x\})]$ Therefore,
$\sqcup \{([[x := a]](\sigma, g))[1] | g \sqsubseteq g^{\#}\} = \cup \{([[x := a]](\sigma, g))[1] | g \subseteq g^{\#}\} =$
$\cup \{\ [(arg(a) \cap g \neq \phi)\ ?\ (g \cup \{x\}) | (g - \{x\})] | g \subseteq g^{\#}\}$

If $arg(a) \cap g^{\#} = \phi$ (thus, also $arg(a) \cap g = \phi$ since $g \sqsubseteq g^{\#}$), then this equals to –
$\cup \{g - \{x\} | g \subseteq g^{\#}\} = g^{\#} - \{x\} = [(\arg(a) \cap g^{\#} \neq \phi)\ ?\ (g^{\#} \cup \{x\})\ |\ (g^{\#} - \{x\})] = [[x := a]]^{\#}(g^{\#})$

Else,
$\cup \{\ldots | g \subsetneq g^{\#}\} \cup (g^{\#} \cup \{x\}) = g^{\#} \cup \{x\} = [(arg(a) \cap g^{\#} \neq \phi)\ ?\ (g^{\#} \cup \{x\}) \mid (g^{\#} - \{x\})] = [[x := a]]^{\#}(g^{\#})$
so in both cases $\sqcup \{([[x := a]](\sigma, g))[1] | g \sqsubseteq g^{\#}\} = [[x := a]]^{\#}(g^{\#})$

In the proof above we showed = and not a $\sqsubseteq$, therefore this abstract interpretation is locally optimal.

The PWhile Programming Language

Here, we define the extension of the While language, called PWhile language, that supports pointers.

We use is the following ***abstract syntax***:

- $a := x \mid *x \mid \&x \mid n \mid a1\ op_a\ a2$
- $b := true \mid false \mid not\ b \mid b1\ op_b\ b2 \mid a1\ op_r\ a2$
- $S := x := a \mid *x := a \mid skip \mid S1\ ;\ S2 \mid if\ b\ then\ S1\ else\ S2 \mid while\ b\ do\ S$

where * and & operators are as in the C language.

The ***concrete semantics*** would be the following:

- Since now a variable can hold an address, let $state: Loc \rightarrow (Loc \cup Z)$, where *Loc* represents a memory address. Denote by *Loc(x)* the location of *x*.
- $[[x := a]](s) = s[loc(x) \rightarrow A[[a]]s]$
- $[[*x := a]](s) = s[s[loc(x)] \rightarrow A[[a]]s]$
- $A[y]s = s[loc(y)]$
- $A[\&y]s = loc(y)$
- $A[*y]s = s[s[loc(y)]]$

For example,

- $[[x := *y]](s) = s[loc(x) \rightarrow s[s[loc(y)]]]$
- $[[*x := y]](s) = s[s[loc(x)] \rightarrow s[loc(y)]]$

Points-to analysis

Since PWhile supports pointers, any (reasonable) analysis of a program would require us to understand the relations between the different pointers. E.g., in constant propagation, we would like to "invalidate" a variable in case the operator * is applied on a pointer that points to it.

We define the ***lattice*** for this analysis as following:

- L = $2^{Var\ X\ Var}$, i.e., a set of variable pairs (in which the first points to the second).
- $pt \sqsubseteq pt' \leftrightarrow pt \subseteq pt'$
- $\sqcup = \cup$, $\sqcap = \cap$
- $\perp = \phi$
- $T = Var \times Var$

And let the abstraction be:

- $\beta(s) = \{(x,y)|s[loc(x)] = loc(y)\}$
- $\alpha(S) = \cup \{\beta(s)|\ s \in S\}$
- $\gamma(pt) = \cup \{s|\ \beta(s) \subseteq pt\}$

The ***abstract semantics*** would be the following:

- Here the state includes the power set of $(Loc \cup Z) \times (Loc \cup Z)$
- $[[x := n]]^{\#}(pt) = pt - \{(x,z)|z \in Z\}$
- $[[x := y]]^{\#}(pt) = pt - \{(x,v)|v \in Var\} \cup \{(x,w)|(y,w) \in pt\}$
- $[[x := \&y]]^{\#}(pt) = pt - \{(x,v)|v \in Var\} \cup \{(x,y)\}$
- $[[x := *y]]^{\#}(pt) = pt - \{(x,v)|v \in Var\} \cup \{(x,w)|\exists_y (y,v),(v,w) \in pt\}$
- $[[*x := y]]^{\#}(pt) = pt - \{(w,v)|\ \exists\ exactly\ one\ (x,w) \in pt\} \cup \{(w,v)|(x,w),(y,v) \in pt\}$
  The "exactly one" requirement comes from the monotonicity and soundness requirements.
  Let's look on the following definition: $[[*x = 0]]^{\#}(pt) = pt - \{(w,z)|(w,x) \in pt, z \in Var\}$
  This is not monotonic, therefore not sound as can be understood from this example:

```
a=&c                    //{(a,c)}
if (...)
        x=&b            //{(a,c),(x,b)}
else
        x=&a            //{(a,c),(x,a)}
                        //{(a,c),(x,a),(x,b)}
*x=null
                        //{(a,c)}
```

  It is easy to see this analysis missed the configuration in which the program will enter the if block, therefore it is not sound.

Now, let's see an example of applying the semantics on a simple program. (We write in the comments the states before and after executing the current line.)

```
/* ϕ */                 t  := &a;               /* {(t, a)}*/
/* {(t, a)}*/          y := &b;                /* {(t, a), (y, b) }*/
/* {(t, a), (y, b)}*/  z := &c;                /* {(t, a), (y, b), (z, c) }*/
                        if x> 0;
                                then p:= &y;    /* {(t, a),  (y, b), (z, c), (p, y)}*/
                                else p:= &z;    /* {(t, a),  (y, b), (z, c), (p, z)}*/
/* {(t, a),  (y, b), (z, c), (p, y), (p, z)}  <= join of the previous two */
                        *p := t;        /* {(t, a), (y, b), (y, c), (p, y), (p, z), (y, a), (z, a)}*/
```

This analysis is sound, but contains many edges, which might not be reachable in run time. This is a conservative analysis, which iterate over all possible running routes (both sides of the if block).
Complexity: $O(n^3)$

<u>Soundness</u>

We need to prove that $[[Stm]]^{\#}(pt) \sqsupseteq \alpha(\{[[Stm]]\sigma|\sigma \in \gamma(\sigma^{\#})\})$, or
$[[Stm]]^{\#}(pt) \supseteq \alpha(\{[[Stm]]\sigma|\sigma \in \gamma(\sigma^{\#})\})$

Let's start with **constant assignment** - x:=a (where a is a constant):

$[[x:=a]]^{\#}(pt) = pt - \{(x,z)|z \in Var\}$

$[[x:=a]](\sigma) = \sigma[loc(x) \to A[[a]]\sigma]$

$\alpha(\{[[x:=a]]\sigma|\sigma \in \gamma(pt)\}) = \alpha(\{\sigma[loc(x) \to a]|\beta(\sigma) \sqsubseteq pt\}) = \sqcup \{\beta(\sigma[loc(x) \to a])|\beta(\sigma) \sqsubseteq pt\}$
$= \cup \{\beta(\sigma[loc(x) \to a])|\beta(\sigma) \sqsubseteq pt\}$

Since *x* points to a constant *a*, there is no $z \in Var$ such that $(x,z) \in \beta(\sigma[loc(x) \to a])$.

Therefore there is no $z \in Var$ such that $(x,z) \in \cup \{\beta(\sigma[loc(x) \to a])|\beta(\sigma) \sqsubseteq pt\}$.

Now we will show that for every $(y,z) \in \cup \{\beta(\sigma[loc(x) \to a])|\beta(\sigma) \sqsubseteq pt\}$, also $(y,z) \in pt$.

If $(y,z) \in \cup \{\beta(\sigma[loc(x) \to a])|\beta(\sigma) \sqsubseteq pt\}$, there exists σ' such that

$(y,z) \in \beta(\sigma'[loc(x) \to a])\ and\ \beta(\sigma') \sqsubseteq pt$.

Since $y \neq x$, according to β's definition $(y,z) \in \beta(\sigma')\ and\ \beta(\sigma') \sqsubseteq pt$, so $(y,z) \in pt$.

And again since $y \neq x$, $(y,z) \in pt - \{(x,w)|w \in Var\} = [[x:=a]]^{\#}(pt)$.

So, $\cup \{\beta(\sigma[loc(x) \to a])|\beta(\sigma) \sqsubseteq pt\} \subseteq [[x:=a]]^{\#}(pt)$, meaning:

$\alpha(\{[[x:=a]]\sigma|\sigma \in \gamma(pt)\}) \sqsubseteq [[x:=a]]^{\#}(pt)$

Assignment into a pointer -

$[[x:=*y]](\sigma) = \sigma[loc(x) \to \sigma[\sigma[loc(y)]]]$

$[[x:=*y]]^{\#}(pt) = (pt - \{(x,z)|z \in Var\}) \cup \{(x,w)|(y,t),(t,w) \in pt\}$

$\alpha(\{[[x:=*y]]\sigma|\sigma \in \gamma(pt)\}) = \alpha(\{\sigma[loc(x) \to \sigma[\sigma[loc(y)]]]|\beta(\sigma) \sqsubseteq pt\}) =$
$= \sqcup \left\{\beta\left(\sigma\left[loc(x) \to \sigma[\sigma[loc(y)]]\right]\right)\middle|\beta(\sigma) \sqsubseteq pt\right\} =$
$= \cup \{\beta(\sigma[loc(x) \to \sigma[\sigma[loc(y)]]])|\beta(\sigma) \sqsubseteq pt\}$

Let $(w,z) \in \cup \{\beta(\sigma[loc(x) \to \sigma[\sigma[loc(y)]]])|\beta(\sigma) \sqsubseteq pt\}$, then there exists σ' such that

$(w,z) \in \beta(\sigma'[loc(x) \to \sigma'[\sigma'[loc(y)]]])$ and $\beta(\sigma') \sqsubseteq pt$.

If $x \neq w$, then like we showed above $(w,z) \in pt - \{(x,u)|u \in Var\}$.

Else, i.e. x=w, $loc(z) = \sigma'[\sigma'[loc(y)]]$ so there is $u \in Var$ such that:

$\sigma'[loc(y)] = loc(u)\ and\ \sigma'[loc(u)] = loc(z)$.

From β's definition, it means that $(u,z),(y,u) \in \beta(\sigma'[loc(x) \to \sigma'[\sigma'[loc(y)]]])$.

Since $y,u \neq x$and $\beta(\sigma') \sqsubseteq pt$ we get that $(u,z),(y,u) \in pt$.

Therefore, $(x,z) \in \{(x,w)|(y,u),(u,w) \in pt\}$, and so

$(x,z) \in (pt - \{(x,z)|z \in Var\}) \cup \{(x,w)|(y,t),(t,w) \in pt\} = [[x:=*y]]^{\#}(pt)$.

From both cases, we get:

$\alpha(\{[[x:=*y]]\sigma|\sigma \in \gamma(pt)\}) = \cup \{\beta(\sigma[loc(x) \to \sigma[\sigma[loc(y)]]])|\beta(\sigma) \sqsubseteq pt\} \subseteq [[x:=*y]]^{\#}(pt)$

The proofs for the rest of the statements are very similar.

Flow insensitive points-to analysis

The previous analysis might be too expensive since many iterations might be needed. A more efficient analysis would be to ignore the control-flow completely, and only accumulate pointers (i.e., no edges, or

points-to relations, are removed during the analysis). Indeed, such analysis is more coarse, but, it allows using more efficient data structure (e.g. for union find) and, thus, getting the analysis in almost a linear time. Space is more efficient as well, since as opposed to the previous analysis, where set of points-to was saved per line, in this analysis one set of points-to is saved per program.

Now, the analysis passes on the statements in a chaotic iteration, and adds new edges to the set of points-to relations, until no edges are added (i.e., no statement changes the points-to set).

Let's see the execution of the analysis for the code we saw earlier:

```
1   t := &a;
2   y := &b;
3   z := &c;
4   if x> 0;
5   then p:= &y;
6         else p:= &z;
7   *p := t;
```

Executed line	State
2	(y,b)
5	(y,b), (p,y)
1	(y,b), (p,y),(t,a)
6	(y,b), (p,y),(t,a),(p,z)
7	(y,b), (p,y),(t,a),(p,z),(y,a), (z,a)
3	(y,b), (p,y),(t,a),(p,z),(y,a), (z,a),(z,c)
...	(y,b), (p,y),(t,a),(p,z),(y,a), (z,a),(z,c)

Another advantage of flow insensitive analysis is straightforward concurrency support (i.e. program with more than one thread).

Usually we cannot have $\alpha(CS) = DF$ on all programs, but we are interested in the precision of the Chaotic iteration lfp on all programs.

The Join-Over-All-Paths (JOP)

Let **paths**(v) denote the potentially infinite set paths from start to v and let **sequence** denote a series of edges upon which the $f^{\#}$ function is activated.
For a sequence of edges $[e_1, e_2, \ldots, e_n]$ define $f^{\#}[e_1, e_2, \ldots, e_n]: L \to L$ by composing the effects on basic blocks $f^{\#}[e_1, e_2, \ldots, e_n](l) = f^{\#}(e_n)(\ldots(f^{\#}(e_2)(f^{\#}(e_1)(l))\ldots)$.
Let **JOP** be the function result on all paths, or more formally:
$JOP[v] = \sqcup \{f^{\#}[e_1, e_2, \ldots, e_n](\iota) \mid [e_1, e_2, \ldots, e_n] \in paths(v)\}$

Let's see the JOP for a constant propagation example:

```
if (x==x):
        x:=2
        y:=3
else:
        x:=3
        y:=2
z: = x+y
```

JOP will return $[x \to \top, y \to \top, z \to 5]$, while the precise answer is $[x \to 2, y \to 3, z \to 5]$. This is caused by the fact that JOP takes into consideration *all* paths, and not only the feasible ones. The good news is that it successfully calculated z=x+y=5. This was possible, due to the JOP's abstract semantics which applied on each path separately and each of them assigned z=5, i.e., $[x \to 2, y \to 3, z \to 5] \sqcup [x \to 3, y \to 2, z \to 5] = [x \to \top, y \to \top, z \to 5]$.

JOP vs. Least Solution

The data-flow (df) solution obtained by Chaotic iteration satisfies for every v: $JOP[v] \sqsubseteq df[v]$. (Proof is done by induction on the path length in which we show that each JOP element is "smaller", i.e. more precise, than the df's element.)

We say that a function $f^{\#}$ is additive (distributive) if $f^{\#}(\sqcup \{x | x \in X\}) = \sqcup \{f^{\#}(x) | x \in X\}$.
If every $f^{\#}_{(u,v)}$ is distributive for all the edges (u,v) then JOP[v]=df[v]. In other words, the JOP is the lfp when combined with the distribute property. That happens because the additive property means that also the iterative analysis will take in account all paths.
(Note that additive function is monotone since $x \sqsubseteq y \Rightarrow x \sqcup y = y \Rightarrow f^{\#}(y) = f^{\#}(x \sqcup y) = f^{\#}(x) \sqcup f^{\#}(y) \sqsupseteq f^{\#}(x)$.)

Few examples

- May be garbage is distributive
  $[[x := exp]](un) = [(arg(exp) \cap un \neq \phi)\ ?\ (un \cup \{x\})\ |\ (un - \{x\})]$
  $un_1 \sqcup un_2$ is distributive.
- Constant propagation is not distributive. Back to the example above,
  - $f_1[cp] = cp[x \to 3, y \to 2]$ (i.e. $f_1(x \to 0, y \to 0) = (x \to 3, y \to 2)$)
  - $f_2[cp] = cp[x \to 2, y \to 3]$
  - $f_3[cp] = cp[z \to x + y]$

  Now we will show they aren't distributive:
  - $f_3(f_1) = [x \to 3, y \to 2, z \to 5]$
  - $f_3(f_2) = [x \to 2, y \to 3, z \to 5]$
  - $f_3(f_1) \sqcup f_3(f_2) = [x \to \top, y \to \top, z \to 5]$ while $f_3(f_1 \sqcup f_2) = [x \to \top, y \to \top, z \to \top]$
- Available expressions is distributive.

- Points-to is not distributive if edges can be removed during the analysis (e.g., due to the destructive operation *x=y we defined earlier). In case edges are only added (like in the flow-insensitive analysis), distributive is achieved.

Complexity of Chaotic Iterations

Usually the complexity depends on the height of the lattice, but in some cases a better bound exists. A function f is called *fast* if $f(f(l)) \sqsubseteq l \sqcup f(l)$. The meaning of “fast” is that there is no need to iterate a loop in the program more than once. For fast functions the Chaotic iterations can be implemented (using DFS) in *O(nest*|V|)* iterations, where *nest* is the number of nested loops and |*V*| is the number of flow control nodes.