1 Reminder

In the previous lesson we discussed the theoretical background which is necessary for abstract interpretation and for the chaotic iteration procedure.

1.1 Monotone

We say that a function f is *monotone* if $a \sqsubseteq b \Rightarrow f(a) \sqsubseteq f(b)$

1.2 Galois Connection

A Galois connection allows us to do a computation in two domains. The idea is that both domains can be arbitrary and one of them is more abstract. The more concrete domain should be defined as $\mathcal{P}(State)$ – the powerset on all states that may occur in a real execution of the program. In this case, we can simplify the definition of abstraction.

We call a pair of functions α and γ that connect two domains A and C ($\alpha: C \to A, \gamma: A \to C$) a Galois connection if:

- $\forall a \in A,\ \alpha(\gamma(a)) \sqsubseteq a$
- $\forall c \in C,\ c \sqsubseteq \gamma(\alpha(c))$
- Both α and γ are monotone

We define the functions for a Galois connection from $\mathcal{P}(State)$ into A in the following way:

- We have a state *abstraction function*, denoted β: State $\to$ A which takes a *concrete state* to an *abstract state*.
- We have an *abstraction function*, denoted α which takes a group of *concrete states* to an *abstract state*.
  $$\alpha(X) = \sqcup \{\beta(\sigma) | \sigma \in X\}$$
  Notice that α is monotone independent of β since if $X \subseteq Y$ then $\alpha(X) \sqsubseteq \alpha(Y)$.
- We have a *concretization function*, denoted γ which takes an *abstract state* to a group of *concrete states*.
  $$\gamma(\sigma^{\#}) = \{\sigma | \beta(\sigma) \sqsubseteq \sigma^{\#}\}$$
  By definition, γ is montone, i.e., if $a \sqsubseteq a'$ then
  $$\gamma(a) = \{\sigma | \beta(\sigma) \sqsubseteq a\} \sqsubseteq \gamma(a') = \{\sigma' | \beta(\sigma') \sqsubseteq a'\}$$
- For the way in which we defined α and γ, it must be that they form a Galois connection as we defined above.

1.3 Local Soundness

We say that a computation is locally sound, if applying each of the statements in the abstract domain, does not yield a more accurate result than abstracting the result of the computation on the concrete domain. Formally

$$[\![Stm]\!]^{\#}(\sigma^{\#}) \sqsupseteq \alpha(\{[\![Stm]\!]\sigma | \sigma \in \gamma(\sigma^{\#})\})$$

1.4 Soundness Theorem

The soundness theorem allows us to say when the fixed-point of a computation is sound (will be defined below). It states that if:

- Let (α, γ) form a Galois connection from C (a concrete domain) to A (an abstract domain)
- Let $f: C \to C$ be a monotone function
- Let $f^{\#}: A \to A$ be a monotone function
- Let any of the following conditions hold:
    - $\forall a \in A: f(\gamma(a)) \sqsubseteq \gamma\left(f^{\#}(a)\right)$
      This condition states that the result of a concrete computation on the concretization of some value, is more accurate than the concretization of the result of the abstract computation. This makes sense, since when doing a computation in the concrete world, we don't lose information, while an abstract computation may lose lots of information each step and therefore its concretization will be less accurate (will have more "valid" results).
    - $\forall c \in C: \alpha(f(c)) \sqsubseteq f^{\#}(\alpha(c))$
      This condition states that the abstraction of the outcome from one step of a concrete computation ($\alpha(f(c))$) is more accurate ($\sqsubseteq$) than doing a corresponding step in the abstract domain over an abstracted input ($f^{\#}(\alpha(c))$). This is because, again, a computation in the abstract domain loses more info than simply abstracting the final result.
    - $\forall a \in A,\ \alpha\left(f(\gamma(a))\right) \sqsubseteq f^{\#}(a)$
      Explanation: $\gamma(a)$ is the group of all values whose abstraction is the same (a) and therefore are undistinguishable in the abstract domain. So, we require that applying an abstraction on the computation on all of them will be at least as accurate as doing the computation in the abstract domain in the first place.

Then the computation of the least-fixed-point is sound, meaning that both of the following properties hold:

- $\mathrm{lfp}(f) \sqsubseteq \gamma\left(\mathrm{lfp}(f^{\#})\right)$
  This means that the concretization of the result of the abstract computation is less accurate than the result of the actual computation. (We denote $\sqsubseteq$ where on the left side we have the more accurate result because it contains fewer options).
- $\alpha(\mathrm{lfp}(f)) \sqsubseteq \mathrm{lfp}(f^{\#})$
  This states that the abstraction of the result of a concrete ("real") computation, is more accurate than the result of doing the entire computation in the abstract domain (which makes sense, like the previous point).

2 Completeness

Completeness means one of two options:

1. $\alpha(\mathrm{lfp}(f)) = \mathrm{lfp}(f^{\#})$
   Computation over the abstract domain is equivalent to doing the computation in the concrete

domain, and then doing an abstraction (i.e., no information is lost for that computation in the abstract domain).

2. $\mathrm{lfp}(f) = \gamma\left(\mathrm{lfp}(f^{\#})\right)$
   The concretization of an abstract computation is equivalent to the computation in the real world (for our specific computation).
   This can happen for example, in a concrete domain of [0,1] and an abstraction which is $Positive$ and $NonPositive$. This isn't pointless, since the abstract computation has a different meaning – it comes to check the property in which we are interested (sign in our case).
   This result simply says that the abstract state describes all the reachable states and these only! It's hard since it means the abstract domain can be used to describe precisely all the reachable parts in the concrete domain (but not necessarily all the concrete domain) – this is generally not the case.

These conditions are **not** equivalent. However, condition number 2 under a Galois Insertion implies condition number 1.

2.1 Proof

Let α and γ form a Galois insertion, meaning $\alpha(\gamma(c)) = c$

Then, start with

$$\mathrm{lfp}(f) = \gamma\left(\mathrm{lfp}(f^{\#})\right)$$

Apply an abstraction on both sides:

$$\alpha(\mathrm{lfp}(f)) = \alpha\left(\gamma\left(\mathrm{lfp}(f^{\#})\right)\right) \xRightarrow{\text{By the Galois Insertion}} \alpha(\mathrm{lfp}(f)) = \mathrm{lfp}(f^{\#})$$

QED ■

3 Constant Propagation

Constant Propagation is another compiler optimization which can be justified via abstract interpretation. This means that we want to track which values have known constant values at some point of the program. We will define the functions for the Galois connection in the following way:

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

$$\beta(\sigma) = \sigma$$

β takes a state (a mapping from variables to $\mathbb{Z}$) to a state in the abstract interpretation world (a mapping from variables to $\mathbb{Z} \cup \{\top, \bot\}$) and in this case (constant propagation) gives the variable the same value.

$$\alpha: \mathcal{P}([Var \to \mathbb{Z}]) \to [Var \to (\mathbb{Z} \cup \{\top, \bot\})]$$

$$\alpha(X) = \sqcup \{\beta(\sigma) | \sigma \in X\} \qquad (= \{\sigma | \sigma \in X\})$$

α is a mapping from a group of concrete states (where each state is a mapping from variables to values), to a single state in the abstract domain, where in this case (constant propagation) each variable is mapped to the most accurate value as possible using a join ($\sqcup$) operator. Practically this means that if a variable is given several constants (by different states in X), it will be given $\top$, otherwise the variable will be given some constant value (if all states agree on one constant or $\bot$). Note that if a variable is given the value $\bot$, it means that the variable was never assigned any value.

$$\gamma(\sigma^{\#}) = \{\sigma | \beta(\sigma) \sqsubseteq \sigma^{\#}\} \qquad (= \{\sigma | \sigma \sqsubseteq \sigma^{\#}\})$$

γ – is mapping from an abstract state, to the group of all possible concrete states that applying an abstraction on them will give back an abstract state which is more accurate than the current one.

3.1 Calculating expressions

The calculation on expressions is defined recursively in the following way:

- For a constant a, $A[\![a]\!]\sigma = a$
- For a variable a, $A[\![x]\!]\sigma = \sigma(x)$
- For a complex operation, we define:
  - For any operation $\mathrm{op} \in \{+, -, *, /\}$ and it's actual meaning op^*
    $$A[\![a \ \mathrm{op} \ b]\!]\sigma = A[\![a]\!]\sigma \ \mathrm{op}^* \ A[\![b]\!]\sigma$$
    - Except for $A[\![0 * x]\!]\sigma$ or $A[\![x * 0]\!]\sigma$ which are both 0

3.2 Calculating abstract expressions

We denote the calculations in the abstract world using $A^{\#}$. We calculate atomic expressions in the same way, but we will change the calculation of non-atomic ("complex") expressions. For complex expressions which only contain constants, nothing is changed. The change will be that $A^{\#}[\![\top \ \mathrm{op} \ b]\!]\sigma^{\#}$and $A^{\#}[\![a \ \mathrm{op} \ \top]\!]\sigma^{\#}$ evaluate to $\top$ (Except for $A^{\#}[\![0 * \top]\!]\sigma^{\#}$ or $A^{\#}[\![x * \top]\!]\sigma^{\#}$ which both evaluate to 0).

The start state is usually one of the following:

- All variables are initialized to 0 (or some other known value)
- All variables are initialized to $\top$ - meaning we expect some unknown random value to be placed in each of the variables (this is often the state in the real world)
- All variables are initialized to $\bot$ - similar to the case with $\top$, this symbolizes that variables begin with some unknown value. The advantage of putting $\bot$ in all of them means that we will be able to detect usage of uninitialized variables and warn about it, in case our analysis discovers that some operation is done where one of the operands is $\bot$.
  In that case, when joining two or more control flows (in the chaotic iteration), if in one of them the value of the variable is $\bot$ then we will define the join to return $\bot$ (unlike the usual join operator that would return the other value!) to symbolize that again the variable may be uninitialized.

3.3 Soundness and Optimality

We say our computation is locally sound:

$$[\![Stm]\!]^{\#}(\sigma^{\#}) \sqsupseteq \alpha(\{[\![Stm]\!](\sigma) | \sigma \in \gamma(\sigma^{\#})\}) = \sqcup \left([\![Stm]\!](\sigma) | \sigma \sqsubseteq \sigma^{\#}\right)$$

This means that an abstract interpretation over an abstract state is less accurate than the abstraction of the computation on an actual state.

3.3.1 Proof of local soundness

The only statement which has any affection on our state is the assignment statement. Obviously, the other statements which do not transform the state (both the abstract and concrete) do not require a proof of soundness.

$$[\![x \coloneqq a]\!]^{\#}(\sigma^{\#}) = \sigma^{\#}[x \to A^{\#}[\![a]\!]\sigma^{\#}], \qquad [\![x \coloneqq a]\!](\sigma) = \sigma[x \to A[\![a]\!]\sigma]$$

We want to prove that

$$[\![x \coloneqq a]\!]^{\#}(\sigma^{\#}) \sqsupseteq\sqcup \left([\![x \coloneqq a]\!](\sigma) \middle| \sigma \sqsubseteq \sigma^{\#}\right)$$

$$\sigma^{\#}[x \to A^{\#}[\![a]\!]\sigma^{\#}] \sqsupseteq\sqcup \left(\sigma[x \to A[\![a]\!]\sigma] \middle| \sigma \sqsubseteq \sigma^{\#}\right)$$

Note that σ is a concrete state (so we can apply A using it) and therefore it has no $\top$ values – only constant values.

For each variable y (where y isn't x) there is a value in $\sigma^{\#}$, and that value can either be some constant or $\top$. If the value is constant, then for all $\sigma \sqsubseteq \sigma^{\#}$ the value of y in σ is the same constant (by the structure of our lattice). If $\sigma^{\#}(y) = \top$, then for all $\sigma \sqsubseteq \sigma^{\#}$, y will evaluate to some constant (can be a different constant in each σ). Applying a join ($\sqcup$) operator on all these constants will yield something, and that will be more accurate than $\top$ (by definition of $\top$) and so, the order of $\sqsupseteq$ will be preserved.

For x, if $A^{\#}[\![a]\!]\sigma^{\#}$ is $\top$, then like the rest of the variables, the $\sqsupseteq$ relation will hold. If $A^{\#}[\![a]\!]\sigma^{\#}$ is a constant, then because of our definition to $A^{\#}$, the only[1] way to get a constant is by using a calculation path which has no $\top$ value in it – and that path is defined exactly using A's definition! Therefore $A[\![a]\!]\sigma = A^{\#}[\![a]\!]\sigma^{\#}$ for all $\sigma \sqsubseteq \sigma^{\#}$

So, we saw that the $\sqsubseteq$ relation holds in all cases for all variables.
Q.E.D. ■

The constant propagation we defined is sound (by Cousot's theorem, local soundness implies global soundness). It also "optimal" when talking about assignments of the form $x \coloneqq a$ (a is a constant or a variable) or $x \coloneqq a \ \text{op} \ b$ where a and b are independent variables or constants. Note that this is an analysis on integers and therefore it does not include division[2].

---

[1] The only way to receive a constant in a computation with $A^{\#}$ that includes a $\top$ value is when computing $0 * \top$ (or $\top * 0$). In that case, $A^{\#}$ will evaluate to 0. In addition, by what we just said, 0 would be computed in a way that A agrees on with $A^{\#}$ and then applying A on a multiplication with it will also yield a zero!
Note that if that 0 is also obtained by a recursion with a case like this, it will eventually reach atomic expressions and from there and upward we can construct this agreement of A and $A^{\#}$.
[2] Note that extending our analysis for division will not be optimal for division statements, since x/x should yield 1 for all x, and we don't do that – if x is $\top$ we return $\top$.

3.3.2 Proof – Local optimality

Since we already proved soundness ($\sqsupseteq$), proving optimality simply requires to prove "accuracy" in the other direction ($\sqsubseteq$). Simply, this means we want to prove that

$$[\![x \coloneqq \exp]\!]^{\#}(\sigma^{\#}) \sqsubseteq \sqcup \left([\![x \coloneqq \exp]\!](\sigma) \middle| \sigma \sqsubseteq \sigma^{\#}\right)$$

$$\sigma^{\#}[x \to A^{\#}[\![\exp]\!]\sigma^{\#}] \sqsubseteq \sqcup \left(\sigma[x \to A[\![\exp]\!]\sigma] \middle| \sigma \sqsubseteq \sigma^{\#}\right)$$

For each variable y (where y isn't x) there is a value in $\sigma^{\#}$, and that value can either be some constant or $\top$. If the value is constant, then for all $\sigma \sqsubseteq \sigma^{\#}$ the value of y in σ is the same constant (by the structure of our lattice). If $\sigma^{\#}(y) = \top$, then for all $\sigma \sqsubseteq \sigma^{\#}$, y will evaluate to some constant (can be a different constant in each σ). Since we can choose two or more different constants (since we are talking about all states σ more accurate than $\sigma^{\#}$), applying a join ($\sqcup$) operator on all these constants will yield $\top$.

So we showed that for each variable y different than x, the value of y on both sides is the same.

For x itself, we need to do the separation:

3.3.2.1 $x \coloneqq a$ (constant or variable)

if $A^{\#}[\![a]\!]\sigma^{\#}$ is a constant, then like the rest of the variables, the $\sqsubseteq$ relation will hold. If $A^{\#}[\![a]\!]\sigma^{\#}$ is $\top$ (only possible when a is a variable), then a may evaluate to any constant value in σ, and so in the join we will get a join on many constants – yielding $\top$ and preserving the $\sqsubseteq$ relation.

3.3.2.2 $x \coloneqq z \ \mathrm{op}\ y$ (for two independent variables/constants)

if $A^{\#}[\![a \ \mathrm{op}\ b]\!]\sigma^{\#}$ is a constant, then like we showed in the soundness proof, there will be an equality for the value of x on both sides of the $\sqsubseteq$ operator.

if $A^{\#}[\![a \ \mathrm{op}\ b]\!]\sigma^{\#}$ evaluates to $\top$, then one of the following holds (note that operators are symmetric when it comes to yielding $\top$):

- $A^{\#}[\![a \ \mathrm{op}\ b]\!]\sigma^{\#}$ where $\mathrm{op} \in \{+, -\}$ and W.L.O.G $A^{\#}[\![a]\!]\sigma^{\#} = \top$
  In that case we can choose $\sigma_1, \sigma_2 \sqsubseteq \sigma^{\#}$ where $A[\![a]\!]\sigma_1 = 1$ and $A[\![a]\!]\sigma_2 = 2$, and so in the join we will receive a join on two different values of $a \ \mathrm{op}\ b$! This means that the result on the right hand-side of the above identity will also be $\top$!
- $A^{\#}[\![a \ \mathrm{op}\ b]\!]\sigma^{\#}$ where $\mathrm{op} \in \{*\}$ and W.L.O.G $A^{\#}[\![a]\!]\sigma^{\#} = \top$ and $A^{\#}[\![b]\!]\sigma^{\#} \neq 0$
  Similar to the previous case…
- Note that $A^{\#}[\![a \ \mathrm{op}\ b]\!]\sigma^{\#} = \top$ where $\mathrm{op} \in \{*\}$ and W.L.O.G $A^{\#}[\![a]\!]\sigma^{\#} = \top$ and $A^{\#}[\![b]\!]\sigma^{\#} = 0$ simply can't happen – because of how we defined $A^{\#}$ on $*$.

<u>Q.E.D. ■</u>

3.3.3 A word about independence

Our analysis isn't always optimal – in fact, it can miss several cases of constants. For example:

$$x - x$$

This would always evaluate to 0 in the concrete computation, but our analysis may give it a $\top$ when $\sigma^{\#}(x) = \top$. There can also be more complex examples.

The point is that dependency between variables introduces a new difficulty which is not trivial to solve in order to achieve optimality, and therefore it's outside the scope of this summary.

3.4 Completness

We say it's not complete, because doing an abstraction over the result of the actual computation does not necessarily yield the same result as doing an abstract computation in the first place. For example:

```
1. Y = 1;
2. X = (1 or 2);
3. If (X % 2 == 0)
4.     Y = 4;
5. Do some stuff without X...
6. If (X % 2 == 1)
7.     Y = 4;
```

Unless our analysis will be smart enough to figure that lines 3 and 5 are opposite conditions that can't change between checks, it will not figure out that Y will always finish as 4 – after line 4 Y will be ⊤ and it will finish like this. So our analysis is not complete.

$$\mathrm{lfp}(f^{\#}) = [x \to \top, y \to \top], \qquad \mathrm{lfp}(f) = \{[x \to 1, y \to 4], [x \to 2, y \to 4]\}$$

And as we can see, both completeness properties **don't** hold:

$$\alpha(\mathrm{lfp}(f)) = [x \to \top, y \to 4] \sqsubset [x \to \top, y \to \top] = \mathrm{lfp}(f^{\#})$$

$$\mathrm{lfp}(f) = \{[x \to 1, y \to 4], [x \to 2, y \to 4]\} \underset{(\subset)}{\sqsubset} \{[x \to a, y \to b] | a, b \in \mathbb{Z}\} = \gamma\left(\mathrm{lfp}(f^{\#})\right)$$

4 Formal available expressions

Here we try to analyze which formal expressions are available at each step and do not require re-computation (this is useful for caching results instead of computing them again). A state is "good" if we have many available expressions, and it's "worse" if it has less available expressions.

4.1 Non-Formal example

Sentence	Expressions available **after** computing the current sentence
$x = y + t$	$\{(y + t)\}$
$z = y + r$	$\{(y + t), (y + r)\}$
while(…) {	$\{(y + t), (y + r)\}$
$t = t + (y + r)$	$\{(y + r)\}$ (note – the original slides had a mistake)
}	

Line 4 requires a bit of explanation, so let's understand what happened there, step by step (a formal explanation is available on the next slide):

- We began with the available expressions from the last step $\{(y + t), (y + r)\}$
- Then we computed a new expression $t + (y + r)$ and added it to the group of available expressions

- Then we assigned a new value for t and by that invalidated all expressions which used the former value of t (including $t + (y + r)$ which we just computed), and therefore we remained only with $\{(y + r)\}$

4.2 Formal definitions

- $\mathcal{P}(Fexp)$
  We denote all the program expressions as $Fexp$, and the domain we are working over is the power set of all the program expressions.
- $X \sqsubseteq Y \Leftrightarrow X \supseteq Y$
  The notation $\sqsubseteq$ which we used previously to denote a state as "more accurate" is now used to define which state has more available expressions. I.E. $X \sqsubseteq Y$ means X is a "better" state than Y since it contains at least all the expressions which were available on Y.
- $X \sqcup Y = X \cap Y$
  The lowest upper bound, which previously meant "the most accurate value that is less accurate than all the given ones" is now the intersection of the available expressions – this is as "good" as a state can be, while still being "worse" than the given ones.
- $\bot = Fexp$
  The "best" state in which we can be, is when all the program expressions (the group $Fexp$) are available
- $\top = \emptyset$
  The worst state is when we have nothing available.

4.3 Finding all the formal expressions

Here we define the computation for our interpretation – this list shows which expressions may be computed (at some potential execution flow) for each statement. This is useful in cases where we want to acquire the list of possible expressions before starting the iteration, for building a finite lattice[3].

- $s ::= \text{skip}$
  Obviously, the skip statement does nothing, and therefore computes nothing ($s.Fexp = \emptyset$).
- $s ::= id := \text{exp}$
  In order to compute an assignment statement, we need to computer the expression which is to be assigned to a variable ($s.Fexp = \text{exp}.rep$[4])
- $s ::= s_1; s_2$
  In order to compute two following sentences, we must compute each of the sentences ($s.Fexp = s_1.Fexp \cup s_2.Fexp$).
- $s ::= \text{while exp do } s_1$
  In order to compute a while statement, we must compute the Boolean expression and the following statement, so the total list of expressions that needs to be computed is the union of the ones for the Boolean condition and the ones for the statement (s.$Fexp = \text{exp}.rep \cup s_1.Fexp$)

[3] So in other words, if we agree to start with an empty lattice and add expressions as we encounter them, we can skip this step of passing over all the statements and collecting expressions
[4] We denote the actual expressions to compute from exp as exp.rep

- $s ::= \text{if exp then } s \text{ else } s$
  Similar to the while statement, we may need to compute the expression and both statements. (s.$Fexp = \text{exp}.rep \cup s_1.Fexp \cup s_2.Fexp$)

4.4 Semantics

We need to explain which expressions are available after each statement. In order to do this formally, we define an instrumented semantics in which a "state" is composed of "variable states" (σ) and "available expressions" (ae). At each step of the computation we begin with a step (σ_0, ae_0) and we define how the current statement Stm transforms that state into (σ_1, ae_1).

Formally, we mark that as

$$S[\![Stm]\!]: State \times \mathcal{P}(Fexp) \to State \times \mathcal{P}(Fexp)$$

For an assignment we denote

$$S[\![id := a]\!](\sigma,\text{ae}) = \big(\sigma[id \to A[\![a]\!]\sigma]\,, notArg(id, ae \cup \{a\})\big)$$

This means that for an assignment statement, the new state is composed out of:

- The old variable state updated with the new value for the variable
- The old list of available expressions combined with the newly computed expression a, and from that we remove all the expressions in which id was an argument. We do that since the assignment to id that was done at the end, invalidated all the expressions that depend on id (id now has a new value which means we need to re-compute them).

For a skip statement, nothing has to be done, so the old state (before executing the skip statement) is the same as the new state. We denote this as

$$S[\![\text{skip}]\!](\sigma,\text{ae}) = (\sigma,\text{ae})$$

After we defined the semantics ($S[\![Stm]\!]$) for each statement which has a meaning other than control flow (assignment and skip), we now need to define the Collecting Semantics. The Collecting Semantics takes a group of states and a statement, and returns a group of possible result states from applying the given statement on any of the states.

$$CS[\![Stm]\!]: \mathcal{P}\big(State \times \mathcal{P}(Fexp)\big) \to \mathcal{P}\big(State \times \mathcal{P}(Fexp)\big)$$

$$CS[\![Stm]\!](X) = \{S[\![Stm]\!](\sigma,\text{ae}) | (\sigma,\text{ae}) \in X\}$$

4.5 CS Example

Let's look at an example of collecting semantics, were we begin with the state

$$([y \to 2, z \to 3, x \to 0, t \to 10, r \to 0], \emptyset)$$

Sentence	Concrete state **after** computing this statement	Abstract**ed** State
	$([y \to 2, z \to 3, x \to 0, t \to 10, r \to 0], \emptyset)$	$\emptyset$
$x = y * z;$	$([y \to 2, z \to 3, x \to 6, t \to 10, r \to 0], \{y * z\})$	$\{y * z\}$
if $(x == 6)$		
$y = z + t;$	$([y \to 13, z \to 3, x \to 6, t \to 10, r \to 0], \{z + t\})$	$\{z + t\}$

…		
if $(x == 6)$		
$r = z + t;$	$([y \to 13, z \to 3, x \to 6, t \to 10, r \to 13], \{z + t\})$	$\{z + t\}$

4.6 Abstract Interpretation

Now we will define the abstract interpretation for Available Expressions like we did with constant propagation:

First we define $\beta: [Var \to \mathbb{Z}] \times \mathrm{P}(\mathrm{Fexp}) \to \mathrm{P}(\mathrm{Fexp})$, the abstract interpretation of a single concrete state (composed of variable states **and** a group of available expressions). The abstract result is a group of available expressions derived from the concrete state:

$$\beta(\sigma, ae) = ae$$

Then we define α:$\mathrm{P}\big([Var \to \mathbb{Z}] \times \mathrm{P}(\mathrm{Fexp})\big) \to \mathrm{P}(\mathrm{Fexp})$, the abstract interpretation of a **group** of concrete states. The abstract result is a group of available expressions derived from all the concrete states together – meaning the expressions **all** of the states agree on:

$$\alpha(\mathrm{X}) = \sqcup\{\beta(\sigma) | \sigma \in \mathrm{X}\} = \cap\{ae | (\sigma, ae) \in \mathrm{X}\}$$

Finally, we define γ:$\mathrm{P}(\mathrm{Fexp}) \to \mathrm{P}\big([Var \to \mathbb{Z}] \times \mathrm{P}(\mathrm{Fexp})\big)$, the concretization of an abstract state. The result is a group of concrete states which their abstract interpretation is "better" (as we defined before) than the given abstract state:

$$\gamma\big(ae^{\#}\big) = \{(\sigma, ae) \big| \beta(\sigma, ae) \sqsubseteq ae^{\#}\} = \{(\sigma, ae) | ae \supseteq ae^{\#}\}$$

4.7 Abstract Semantics

Now we will define the abstract semantics for Available Expressions.

For each statement, we show how it affects the state in the abstract interpretation:

$$\mathrm{S}[\![Stm]\!]^{\#}\colon\ \mathrm{P}(\mathrm{Fexp}) \to \mathrm{P}(\mathrm{Fexp})$$

- For the skip statement the available expressions stay the same:
$$\mathrm{S}[\![skip]\!]^{\#}(ae) = \mathrm{ae}$$
- For an assignment we add the calculated expression, and then remove all expressions containing the variable we've just changed (because the assignment has just invalidated them):
$$\mathrm{S}[\![id := a]\!]^{\#}\big(ae^{\#}\big) = notArg(id\,, ae^{\#} \cup \{a\})$$
- All other statements in the *while* language are irrelevant because they determine only the control flow of the program and have no actual meaning. These will only matter for determining the "neighbors" of each statement when doing the Chaotic Iteration, as we did in the previous lessons.

4.8 Example

Now, let's see an example abstract interpretation (for the previous program):

Sentence	Abstract state after the current statement	Explanation
	$\emptyset$	We begin with nothing computed

$x = y * z;$	$\{y * z\}$	
$\text{if } (x == 6)$		We now begin an optional execution branch
$y = z + t;$	$\{z + t\}$	
	$\{y * z\} \sqcup \{z + t\} = \emptyset$	When finishing the branch, merge it back with the main using a join operator
…		
$\text{if } (x == 6)$		
$r = z + t;$	$\{z + t\}$	We now begin an optional execution branch

Note that the Collecting Semantics is much more accurate (in the CS, at each state we had at least the same expressions available if not more) than our analysis – our analysis will only compute available expressions (unlike the CS which also tracks the variable states) and so it won't know that we will definitely enter the first "if" statement, making the expression $z + t$ available in the second "if" statement. In the AI, we can see that we had an $\emptyset$ of available expressions, while in the CS (1-2 pages ago) exactly at the same place, we had one available expression! Since our AI will miss an available expression, it is **not** complete!

4.9 Properties of the interpretation:

- Local Soundness:

$$[\![st]\!]^{\#}(ae^{\#}) \sqsupseteq \sqcup \{([\![st]\!](\sigma,\text{ae}))[1] \mid \text{ae} \sqsubseteq \text{ae}^{\#}\}$$
$$[\![st]\!]^{\#}(ae^{\#}) \subseteq \cap \{([\![st]\!](\sigma,\text{ae}))[1] \mid \text{ae} \supseteq \text{ae}^{\#}\}$$

  Note that $[\![st]\!](\sigma,\text{ae}) = (\sigma_1, \text{ae}_1)$ and the $[1]$ at the end is like an array reference, which is meant to return ae_1

4.9.1 Proof

We will prove the local soundness for all types of statements for which we defined some meaning:

$$\text{S}[\![skip]\!]^{\#}(ae^{\#}) = \text{ae}^{\#}$$

$$S[\![\text{skip}]\!](\sigma,\text{ae}) = (\sigma,\text{ae})$$

$$\cap \{([\![skip]\!](\sigma,\text{ae}))[1] \mid \text{ae} \supseteq \text{ae}^{\#}\} = \cap \{(\sigma,\text{ae})[1] \mid \text{ae} \supseteq \text{ae}^{\#}\}$$
$$= \cap \{\text{ae} \mid \text{ae} \supseteq \text{ae}^{\#}\} \overset{*}{=} \text{ae}^{\#} \underset{\sqsubseteq}{=} [\![skip]\!]^{\#}(ae^{\#})$$

* As we can see, in the intersection we can take $\text{ae} \equiv \text{ae}^{\#}$ (since $\text{ae}^{\#} \supseteq \text{ae}^{\#}$) and then we intersect them all, we get the smallest group which is in fact $\text{ae}^{\#}$.

$$\text{S}[\![id := a]\!]^{\#}(ae^{\#}) = notArg(id\,, ae^{\#} \cup \{a\})$$

$$S[\![id := a]\!](\sigma,\text{ae}) = (\sigma[id \to A[\![a]\!]\sigma]\,, notArg(id, ae \cup \{a\}))$$

First of all let's show that $notArg$ is monotone (in the expression list). Let ae_1 and ae_2 be two expression lists, where $\text{ae}_1 \subseteq \text{ae}_2$. So

$$notArg(id, \text{ae}_2) = notArg(id, \text{ae}_1 \cup (ae_2 - ae_1)) = notArg(id, \text{ae}_1) \cup notArg(id, ae_2 - ae_1)$$
$$\supseteq notArg(id, \text{ae}_1)$$

$$\cap\left\{\left(\llbracket id \coloneqq a\rrbracket(\sigma,\mathrm{ae})\right)[1] \middle| \mathrm{ae} \supseteq \mathrm{ae}^{\#}\right\} = \cap\left\{\left(\left(\sigma[id \to A\llbracket a\rrbracket\sigma], notArg(id, ae \cup \{a\})\right)\right)[1] \middle| \mathrm{ae} \supseteq \mathrm{ae}^{\#}\right\}$$

$$= \{notArg(id, ae \cup \{a\}) | \mathrm{ae} \supseteq \mathrm{ae}^{\#}\} \underset{\substack{\text{By monotonicity of } notArg \\ \text{we take the "smallest" elemnt}}}{=} notArg(id, ae^{\#} \cup \{a\})$$

$$= \mathrm{S}\llbracket id \coloneqq a\rrbracket^{\#}(ae^{\#})$$

We proved local soundness for all statements for which we defined an interpretation (for statements such as while, for which we defined no interpretation, it means that in the AI they serve as identity transforms), and therefore following Cousot's theorem we gain global soundness. ■

- Optimality

  $$\llbracket st\rrbracket^{\#}(ae^{\#}) = \{\alpha(\llbracket st\rrbracket(\sigma,\mathrm{ae})) | \sigma \in \gamma(\sigma^{\#})\} = \sqcup\{(\llbracket st\rrbracket(\sigma,\mathrm{ae}))[1] | \sigma \sqsubseteq \sigma^{\#}\}$$

  In the proof of local soundness, we had an equality and not a $\sqsubseteq$, meaning that our AI is indeed locally optimal. ■

5 May-Be-Garbage Analysis

- A variable x may-be-garbage at a program point *v* if there exists an execution path leading to *v* which x's value is unpredictable. We say that a value is unpredictable if one of the following conditions hold:
  (1) Was not assigned
  (2) Was assigned using an unpredictable expression
- We will define the Lattice as following:
  - $L_{mgb} = \mathcal{P}(Var)$
    The group of all **unpredictable** variables.
  - $\sqsubseteq\ =\ \subseteq$
    A state is "more accurate" if less variables are unknown, so we define "more accurate"="list of unknown variables is contained in the other one"
  - $\perp = \emptyset$
    The most accurate state is when no variables are unknown
  - $\top = Var$
    The least accurate state is when all variables are unknown
  - $\sqcup = \cup$
    The most accurate state which is less accurate than two given states, is when the group of unknown variables is the union of the unknown groups of each state
  - $\sqcap = \cap$
    Similar to $\sqcup$=$\cup$
  - Initial state is Var
    All variables are NOT assigned
- The abstract interpretation:
  - $\beta: [Var \to \mathbb{Z}] \times \mathrm{P}(Var) \to \mathrm{P}(Var)$
    $\beta(\sigma, gar) = gar$
  - α and γ are defined accordingly.
- The abstract semantics: [arg(exp) means the arguments in the expression)
  - $\llbracket x \coloneqq exp\rrbracket^{\#}(gar^{\#}) = \begin{cases} gar^{\#} - \{x\}, & if\ \arg(exp) \cap gar^{\#} = \emptyset \\ gar^{\#} \cup \{x\}, & else \end{cases}$

5.1 Properties

- Galois Connection

5.1.1 Proof

Let a_{gar} be a group of variables (representing the may-be-garbage variables)

$$\alpha\left(\gamma(a_{gar})\right) = \alpha\left(\left\{(\sigma, a'_{gar}) \middle| \underbrace{\beta(\sigma, a'_{gar})}_{a'_{gar}} \sqsubseteq a_{gar}\right\}\right) = \sqcup\left\{\underbrace{\beta(\sigma, a'_{gar})}_{a'_{gar}} \middle| a'_{gar} \sqsubseteq a_{gar}\right\} \stackrel{\text{def}}{=} a_{gar}$$
$$\Rightarrow \alpha\left(\gamma(a_{gar})\right) \sqsubseteq a_{gar}$$

Let $c_{gar} = \{(\sigma_i, a_{gar_i})\}_{i=1}^{k}$ be a group of concrete states

$$\gamma(\alpha(c)) = \gamma\left(\alpha\left(\{(\sigma_i, a_{gar_i})\}_{i=1}^{k}\right)\right) = \gamma\left(\bigsqcup_{i=1}^{k} \underbrace{\beta\left(\sigma_i, a_{gar_i}\right)}_{a_{gar_i}}\right)$$
$$= \left\{(\sigma, a'_{gar}) \middle| \forall\sigma, \underbrace{\beta(\sigma, a'_{gar})}_{a'_{gar}} \sqsubseteq \bigsqcup_{i=1}^{k} a_{gar_i}\right\} = \left\{(\sigma, a'_{gar}) \middle| \forall\sigma, a'_{gar} \sqsubseteq \bigsqcup_{i=1}^{k} a_{gar_i}\right\}$$
$$= \left\{(\sigma, a'_{gar}) \middle| \forall\sigma, a'_{gar} \subseteq \bigcup_{i=1}^{k} a_{gar_i}\right\} \supseteq \{(\sigma_i, a_{gar_i})\}_{i=1}^{k} \Rightarrow c \sqsubseteq \gamma(\alpha(c))$$

We can easily see that α and γ are monotone, since β does not change the state – it only takes its second field.

- Soundness

5.1.2 Proof

$$[\![x := exp]\!](\sigma, gar) = \left(\sigma[x \to A[\![exp]\!]\sigma], \begin{cases} gar - \{x\}, & if\ \arg(exp) \cap gar = \emptyset \\ gar \cup \{x\}, & else \end{cases}\right)$$

$$[\![x := exp]\!]^{\#}(gar^{\#}) = \begin{cases} gar^{\#} - \{x\}, & if\ \arg(exp) \cap gar^{\#} = \emptyset \\ gar^{\#} \cup \{x\}, & else \end{cases}$$

$$\sqcup\left\{\left([\![x := exp]\!](\sigma, \mathrm{gar})\right)[1] \middle| \mathrm{gar} \sqsubseteq \mathrm{gar}^{\#}\right\} = \cup\left\{\left([\![x := exp]\!](\sigma, \mathrm{gar})\right)[1] \middle| \mathrm{gar} \subseteq \mathrm{gar}^{\#}\right\}$$
$$= \cup\left\{\begin{cases} gar - \{x\}, & if\ \arg(exp) \cap gar = \emptyset \\ gar \cup \{x\}, & else \end{cases} \middle| \mathrm{gar} \subseteq \mathrm{gar}^{\#}\right\} =$$
$$= \cup\left\{\begin{cases} gar - \{x\}, & if\ \arg(exp) \cap gar = \emptyset \\ gar \cup \{x\}, & else \end{cases} \middle| \mathrm{gar} \subseteq \mathrm{gar}^{\#}\right\} = (\$)$$

<u>Case 1</u>: $\arg(exp) \cap gar^{\#} = \emptyset$ (which implies $\arg(exp) \cap gar = \emptyset$ for $\mathrm{gar} \subseteq \mathrm{gar}^{\#}$)

$$(\$) = \cup\{gar - \{x\} | \mathrm{gar} \subseteq \mathrm{gar}^{\#}\} = \mathrm{gar}^{\#} - \{x\} = \begin{cases} gar^{\#} - \{x\}, & if\ \arg(exp) \cap gar^{\#} = \emptyset \\ gar^{\#} \cup \{x\}, & else \end{cases}$$
$$= [\![x := exp]\!]^{\#}(gar^{\#})$$

<u>Case 2</u>: Else (not case 1) (which implies $gar^{\#} \cup \{x\}$ will be in the union, when $gar = gar^{\#}$)

$$(\$) = \cup \underbrace{\{\ldots \mid \text{gar} \subsetneq \text{gar}^{\#}\}}_{\subseteq gar^{\#} \cup \{x\}} \cup \left(gar^{\#} \cup \{x\}\right) = gar^{\#} \cup \{x\} = \begin{cases} gar^{\#} - \{x\}, & if\ \arg(exp) \cap gar^{\#} = \emptyset \\ gar^{\#} \cup \{x\}, & else \end{cases}$$

$$= [\![x := exp]\!]^{\#}\left(gar^{\#}\right)$$

We can see that in both cases $\sqcup \left\{\left([\![x := exp]\!](\sigma, \text{gar})\right)[1] \middle| \text{gar} \sqsubseteq \text{gar}^{\#}\right\} \underset{\sqsubseteq}{=} [\![x := exp]\!]^{\#}\left(gar^{\#}\right)$. ■

- Optimality – In the proof of local soundness, we had an equality and not a $\sqsubseteq$, meaning that our AI is indeed locally optimal. ■
- Not complete
  - Similarly to the constant propagation, not tracking concrete values can miss control flows that will initialize things

6 The PWhile programming language (while language with pointers)

We will begin by defining the syntax of the new language

$exp ::= x \mid *x \mid \&x \mid n \mid exp_1\ op_{exp}\ exp_2$

$bool ::= \text{true} \mid \text{false} \mid \text{not}\ bool \mid bool_1 op_{bool} bool_2$

$stmt ::= x := exp \mid *x := \text{exp} \mid \text{skip} \mid stmt_1; stmt_2 \mid$

if $bool$ **then** $stmt_1$ **else** $stmt_2$ | **while** $bool$ **do** $stmt$

6.1 Concrete semantics for PWhile

In PWhile, we changed our previous definition of states. Instead of having a mapping from a variable to its value, we define a mapping from a **location** (which corresponds to the (memory) location of that variable) to another location or (integer) value:

$$state: Loc \to Loc \cup \mathbb{Z}$$

For every atomic statement S we define the semantic function:

$$[\![S]\!]: state \to state$$

- We define loc(x) as the location of variable x.
- $[\![x := a]\!](\sigma) = \sigma[\text{loc}(x) \to \text{A}[\![a]\!]\sigma]$ where:
  - For a constant n, $\text{A}[\![n]\!]\sigma = n$
  - For a variable y, $\text{A}[\![y]\!]\sigma = \sigma\left(\text{loc}(y)\right)$
  - For a variable y, $\text{A}[\![\&y]\!]\sigma = \text{loc}(y)$
    Meaning in an assignment the location of x will change to the value in σ of the location of y. (it is possible its value is another location)
  - For a variable y, $\text{A}[\![*y]\!]\sigma = \sigma\left(\sigma\left(\text{loc}(y)\right)\right)$
    Meaning in an assignment the location of x will change to the value of the value in σ of the location of y.
  - Intuition: &y is the address (location) of y, y is the value of y, and *y is the value of y's inner address.
    Meaning in an assignment the location of x will change to the location of y.
  - $\text{A}[\![exp_1\ op_{exp}\ exp_2]\!]\sigma = \text{A}[\![exp_1]\!]\sigma\ op_{exp}\ \text{A}[\![exp_2]\!]\sigma$

- $[\![*x \coloneqq a]\!](\sigma) = \sigma[\sigma(\mathrm{loc}(x)) \to \mathrm{A}[\![a]\!]\sigma]$
  Meaning the inner value of x (the value of the location it points to) will change to the semantics of a (as defined in A).

7 Points-To-Analysis - Which Variables points to which variable

7.1 What do we need Points-To Analysis for?

Basically, we need this for every analysis on a language which supports pointers. For example, if in C we would like to do constant propagation, we would want to know when assigning through pointers, which variables may be affected. Since our points-to analysis may produce more pointers than the actual pointing-state, whenever an assignment through a pointer does not agree with the previous value, we will put a ⊤ on that variable. This will allow us to keep the soundness of the constant propagation, since we won't generate more constants than any concrete run.

7.2 Formal definition

The point of this analysis is not to miss any points-to relation between two variables (such as x points at y) and we define a state as "more accurate"/"better" if it has **"less"** point-to relations.

- We will define the Lattice L_{pt} as following:
  - L = $\mathrm{P}(Var \times Var)$
    The lattice is composed of the group of all variable pairs (the first one points to the second)
  - $\sqsubseteq\ =\ \subseteq$
    A state is "more accurate" if it's group of point-to pairs is contained inside the group of the other state.
  - $\top = Var \times Var$
    When we don't know anything, assume all variables are pointing at all the others, since we don't want to miss any points-to pair.
  - $\bot = \emptyset$
    The most accurate state in this analysis (where we don't want to miss any point-to pair) is when we know nothing points at nothing.
  - $\sqcup = \cup$
    When we need to find the most accurate state which is less accurate than all given states, create a union of the points-to pairs, to make sure we are least accurate than all given states
  - $\sqcap = \cap$
    Similar logic to ⊔, but reversed
- Abstraction:
  $$\beta: (Loc \to Loc \cup \mathbb{Z}) \to \mathrm{P}(Var \times Var)$$
  The β function takes a state σ and returns the group of all pairs of variables <*x,y*>, which the value of the location of *x* is the location of *y* (meaning that x points to y):
  $$\beta(\sigma) = \{(x, y) | \sigma(\mathrm{loc}(x)) = \mathrm{loc}(y)\}$$

The α function takes a group of states and returns the union of the abstraction of each state (meaning all the "variable points-to" option that exists in them):

$$\alpha: \mathcal{P}(Loc \to Loc \cup \mathbb{Z}) \to \mathrm{P}(Var \times Var)$$
$$\alpha(A) = \cup \{(x,y) | \sigma \in A \wedge \sigma(\mathrm{loc}(x)) = \mathrm{loc}(y)\}$$

7.3 Abstract Semantics

Let $\sigma^{\#}$ be an abstract state (i.e., a group of point-to pairs). We will now analyze the abstract semantics for assignment statements:

- $[\![\boldsymbol{x} := a]\!]^{\#}(pt) = pt - \{(\boldsymbol{x}, \boldsymbol{z}) | \boldsymbol{z} \in \boldsymbol{Var}\}$
  If we assign a constant a to a variable x, remove all the pairs in which x points at some other variable
- $[\![\boldsymbol{x} := \boldsymbol{y}]\!]^{\#}(pt) = (pt - \{(\boldsymbol{x}, \boldsymbol{z}) | \boldsymbol{z} \in \boldsymbol{Var}\}) \cup \{(\boldsymbol{x}, \boldsymbol{w}) | (\boldsymbol{y}, \boldsymbol{w}) \in \boldsymbol{pt}\}$
  If we assign the value of a variable y to a variable x, like before remove all the pairs in which x points at some other variable, and for each pair that y points at w, add a pair of x pointing at w
- $[\![\boldsymbol{x} := \&\boldsymbol{y}]\!]^{\#}(pt) = (pt - \{(\boldsymbol{x}, \boldsymbol{z}) | \boldsymbol{z} \in \boldsymbol{Var}\}) \cup \{(\boldsymbol{x}, \boldsymbol{y})\}$
  If we assign the location of the variable y to a variable x, remove all the pairs in which x points at some other variable, and add a pair saying that x points to y
- $[\![\boldsymbol{x} := *\boldsymbol{y}]\!]^{\#}(pt) = (pt - \{(\boldsymbol{x}, \boldsymbol{z}) | \boldsymbol{z} \in \boldsymbol{Var}\}) \cup \{(\boldsymbol{x}, \boldsymbol{w}) | (\boldsymbol{y}, \boldsymbol{t}), (\boldsymbol{t}, \boldsymbol{w}) \in \boldsymbol{pt}\}$
  If we assign the value pointed by a variable y to a variable x, like before remove all the pairs in which x points at some other variable, and for each pair that t points at w and y points at t, add a pair of x pointing at w
- $[\![*\boldsymbol{x} := \boldsymbol{y}]\!]^{\#}(pt) = (pt - \{(\boldsymbol{w}, \boldsymbol{p}) | \boldsymbol{p} \in \boldsymbol{Var}, (\boldsymbol{x}, \boldsymbol{w}) \in \boldsymbol{pt} \text{ and } \forall \boldsymbol{z}, (\boldsymbol{x}, \boldsymbol{z}) \in \boldsymbol{pt} \Rightarrow \boldsymbol{z} = \boldsymbol{w}\}) \cup \{(\boldsymbol{w}, \boldsymbol{p}) | (\boldsymbol{x}, \boldsymbol{w}), (\boldsymbol{y}, \boldsymbol{p}) \in pt\}$
  If we assign the value of a variable y to the variable pointed by x (we mark the variable pointed by x as w), if x indeed points at most at one variable w (This is the condition $\forall z, (x,z) \in pt \Rightarrow z = w$) then remove all the pairs in which w points at some other variable. Then for each pair that y points at p, add a pair of w pointing at p.
  It wouldn't be sound to delete the group of pointers of the form (w,p) when x points at more than one variable w. Why? We'll demonstrate below.
- Continuing $[\![*x := *y]\!]$, $[\![*x := a]\!]$ and $[\![*x := \&y]\!]$ is done in a similar way

7.4 Example for un-soundness in $[\![*\boldsymbol{x} := \boldsymbol{y}]\!]$

Why would it be unsound to delete the pointer group

$$\{(\boldsymbol{w}, \boldsymbol{p}) | \boldsymbol{p} \in \boldsymbol{Var}, (\boldsymbol{x}, \boldsymbol{w}) \in \boldsymbol{pt}\}$$

when x points at more than one variable in our analysis?

Let's begin with some concrete state in which $x > 0$ on the following example

Sentence	state **after** the current statement	
	Concrete	Abstract
	$\emptyset$	$\emptyset$
`y := &b;`	$\{(y,b)\}$	$\{(y,b)\}$
`z := &c;`	$\{(y,b),(z,c)\}$	$\{(y,b),(z,c)\}$
`if x>0`		
`then x:= &y;`	$\{(y,b),(z,c),(x,y)\}$	$\{(y,b),(z,c),(x,y)\}$

`else x:= &z;`	Not executed	$\{(y,b),(z,c),(x,z)\}$
	$\{(y,b),(z,c),(x,y)\}$	$\{(y,b),(z,c),(x,y)\} \sqcup \{(y,b),(z,c),(x,z)\}$ $= \{(y,b),(z,c),(x,y),(x,z)\}$
`*x := &t;`	$\{(y,t),(z,c),(x,y),(x,z)\}$	$(\{(y,b),(z,c),(x,y),(x,z)\} - \{(\boldsymbol{y},\boldsymbol{b}),(\boldsymbol{z},\boldsymbol{c})\})$ $\cup \{(y,t),(z,t)\}$ $= \{(y,t),(z,t),(x,y),(x,z)\}$

We defined the accuracy of our analysis by forcing it to include at least all the pointers that the concrete computation will have, but it missed the point-to pair (z,c)! Therefore, it's not sound.

7.5 Good Example with correct semantics

Sentence	Abstract state **after** the current statement	Explanation
	$\emptyset$	We begin with nothing pointing at anything
`t := &a;`	$\{(t,a)\}$	
`y := &b;`	$\{(t,a),(y,b)\}$	
`z := &c;`	$\{(t,a),(y,b),(z,c)\}$	
`if x>0`		
`then p:= &y;`	$\{(t,a),(y,b),(z,c),(p,y)\}$	Potential execution #1
`else p:= &z;`	$\{(t,a),(y,b),(z,c),(p,z)\}$	Potential execution #2
	$\{(t,a),(y,b),(z,c),(p,y)\}$ $\sqcup \{(t,a),(y,b),(z,c),(p,z)\}$ $= \{(t,a),(y,b),(z,c),(p,y),(p,z)\}$	When finishing all potential executions, merge to a main branch using a join operator
`*p := t;`	$\{(t,a),(y,b),(z,c),(p,y),(p,z),(y,a),(z,a)\}$	p pointed at more than one variable (z and y), so we did not remove the pairs of the form $(z,\dots)$ or $(y,\dots)$

7.6 Properties

- Galois Connection

7.6.1 Proof

Let a_{pt} be a group of variable pairs (representing the potential points-to relations)

$$\alpha\left(\gamma(a_{pt})\right) = \alpha(\{\sigma | \beta(\sigma) \sqsubseteq a_{pt}\})$$

We will mark $\{(x,y) | \sigma(\mathrm{loc}(x)) = \mathrm{loc}(y)\} = \beta(\sigma) = a'_{pt}$. Since σ was any state for which it's abstraction was more accurate than a_{pt} (which means it's subgroup of variables pointing one to another was contained in a_{pt}), we can have such a σ so that $a'_{pt} = a_{pt}$.

$$= \alpha(\{\sigma | a'_{pt} \sqsubseteq a_{pt}\}) = \sqcup \left\{ \underbrace{\beta(\sigma)}_{a'_{pt}} \middle| a'_{pt} \sqsubseteq a_{pt} \right\} = \cup \{a'_{pt} | a'_{pt} \subseteq a_{pt}\} = a_{pt} \Rightarrow \alpha\left(\gamma(a_{pt})\right) \sqsubseteq a_{pt}$$

Let $c_{pt} = \{\sigma_i\}_{i=1}^{k}$ be a group of concrete states

$$\gamma\left(\alpha(c_{pt})\right) = \gamma\left(\alpha(\{\sigma_i\}_{i=1}^{k})\right) = \gamma\left(\bigsqcup_{i=1}^{k} \beta(\sigma_i)\right)$$

We will mark $\{(x,y)|\sigma_i(\mathrm{loc}(x)) = \mathrm{loc}(y)\} = \beta(\sigma_i) = a_{pt_i}$ where the same properties hold as in the previous definition.

$$= \gamma\left(\bigsqcup_{i=1}^{k} a_{pt_i}\right) = \left\{\sigma' \middle| \beta(\sigma') \sqsubseteq \bigsqcup_{i=1}^{k} a_{pt_i}\right\} = \left\{\sigma' \middle| \beta(\sigma') \subseteq \bigcup_{i=1}^{k} a_{pt_i}\right\}$$

For each i, we can choose $\sigma' = \sigma_i$ and then $\beta(\sigma') = a_{pt_i} \overset{\text{def}}{\subseteq} \bigcup_{i=1}^{k} a_{pt_i}$ and so we have σ_i in that group. From here we can see that

$$c_{pt} \subseteq \left\{\sigma' \middle| \beta(\sigma') \subseteq \bigcup_{i=1}^{k} a_{pt_i}\right\} = \gamma\left(\alpha(c_{pt})\right) \Rightarrow c_{pt} \sqsubseteq \gamma\left(\alpha(c_{pt})\right)$$

We can easily see that α and γ are monotone; α is monotone because of its union-like operation, and γ is monotone because of a similar reason. ■

- Soundness

7.6.2 Proof

We will prove only 2 out of 8 statements, the rest are very similar. What we want to prove is

$$[\![Stm]\!]^{\#}(pt) \sqsupseteq \alpha(\{[\![Stm]\!]\sigma | \sigma \in \gamma(\sigma^{\#})\})$$

Or simply

$$[\![Stm]\!]^{\#}(pt) \supseteq \alpha(\{[\![Stm]\!]\sigma | \sigma \in \gamma(\sigma^{\#})\})$$

Let's begin

7.6.2.1 Constant assignment

$[\![x := a]\!](\sigma) = \sigma[\mathrm{loc}(x) \to \mathrm{A}[\![a]\!]\sigma,$ $\quad [\![x := a]\!]^{\#}(pt) = pt - \{(x,z)|z \in Var\},$ $\quad$ (a is a const)

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

We know that $\beta(\sigma[loc(x) \to a])$ does not contain any pair of the form (x,z) because x points at a constant a and therefore cannot point at a variable. So the union ($) does not contain any pair of the form (x,z).

Now, let's assume $(y,z) \in (\$)$ and show that $(y,z) \in pt$. If $(y,z) \in (\$)$, there exists a σ' so that

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

Because $y \neq x$ we can see that according to β's definition

$$(y,z) \in \beta(\sigma') \text{ and } \beta(\sigma') \sqsubseteq pt$$

$$(y,z) \in pt$$

Now, since $(y,z) \in (\$)$, $y \neq x$ and so $(y,z) \in pt - \{(x,w)|w \in Var\} \underline{= [\![x := a]\!]^{\#}(pt)}$

In total, we got

$$(\$) \subseteq [\![x := a]\!]^{\#}(pt)$$

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

7.6.2.2 Assignment into pointer

$$[\![x := {}^*y]\!](\sigma) = \sigma\left[\mathrm{loc}(x) \to \sigma\left(\sigma(\mathrm{loc}(y))\right)\right]$$

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

$$\begin{aligned}\alpha(\{[\![x := {}^*y]\!]\sigma | \sigma \in \gamma(pt)\}) &= \alpha\left(\left\{\sigma\left[loc(x) \to \sigma\left(\sigma(\mathrm{loc}(y))\right)\right] \middle| \beta(\sigma) \sqsubseteq pt\right\}\right)\\ &= \sqcup \left\{\beta\left(\sigma\left[loc(x) \to \sigma\left(\sigma(\mathrm{loc}(y))\right)\right]\right) \middle| \beta(\sigma) \sqsubseteq pt\right\} =\\ &= \cup \left\{\beta\left(\sigma\left[loc(x) \to \sigma\left(\sigma(\mathrm{loc}(y))\right)\right]\right) \middle| \beta(\sigma) \subseteq pt\right\} = (\$)\end{aligned}$$

Let (w,z) be a pair in ($) → there exists a σ' so that $(w,z) \in \beta\left(\sigma'\left[loc(x) \to \sigma'\left(\sigma'(\mathrm{loc}(y))\right)\right]\right)$

<u>Case 1 - $w \neq x$</u>

Then like previously, we can see that $(w,z) \in pt - \{(x,u) | u \in Var\}$

<u>Case 2 - $w = x$ (the pair is now (x,z))</u>

Then obviously $\sigma'\left(\sigma'(\mathrm{loc}(y))\right) = loc(z)$, so there exists some variable u so that:

$$\sigma'(\mathrm{loc}(y)) = loc(u), \qquad \sigma'(loc(u)) = loc(z)$$

In that case, by β's definition, (u,z) and (y,u) will be in $\beta\left(\sigma'\left[loc(x) \to \sigma'\left(\sigma'(\mathrm{loc}(y))\right)\right]\right)$, and since $u, y \neq x$ then these relations also exist in $\beta(\sigma')$ which is included in pt.

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

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

So, from both cases we get

$$\alpha(\{[\![x := {}^*y]\!]\sigma | \sigma \in \gamma(pt)\}) = (\$) \subseteq [\![x := {}^*y]\!]^{\#}(pt)$$

8 Flow-Insensitive Points-To Analysis

Our analysis of points-to was potentially expensive, since we need to check the neighbor nodes (in the chaotic iteration) and potentially do many iterations in different branches where for each one we need to compute the union of the two sets.

Another approach for the points-to analysis works by ignoring the control-flow and saying that all statements are "neighbors" in the program flow graph. In order to make our analysis sound with such random execution, we never remove any pairs from the set of point-to pairs – we just keep

iterating over all the sentences in the program until we reach a fixed-point (i.e., passing on any sentence now will not make any difference to the group of points-to pairs).

Since we only keep one set of point-to pairs, and we never remove pairs, we can use a union-find data structure, which allows us to add pairs and check for their existence in almost linear time (amortized inverse Ackerman if we want to be exact).

8.1 Example

Sentence	Abstract state after the current statement
	$\emptyset$
`t := &a;`	$\{(t,a)\}$
`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;`	$\{(t,a),(y,b),(z,c),(p,y),(p,z)\}$
`*p := t;`	$\{(t,a),(y,b),(z,c),(p,y),(p,z),(y,a),(z,a)\}$

The final point-to set is

$$\{(t,a),(y,b),(z,c),(p,y),(p,z),(y,a),(z,a)\}$$

You can see that if we re-iterate with it, it will not change – so this will be the set of point-to pairs that we will use for **all the locations in the program** (since we ignored control-flow we can't say that some are only valid in some place).