ETH Zürich
Chair of Software Engineering

Software Verification

Bertrand Meyer

The alias calculus

Note

These slides describe the alias calculus as of 2010. The core concepts remain but a better mathematical model is currently used. See recent publications on the topic.

Hoare-style reasoning

Assignment rule:

{P (e)} x := e {P (x)}

Hoare-style reasoning

```
require
        y + 1 < 3
do
            -- y + 1 < 3
        x := whatever + 10000
             -- y + 1 < 3
        y := y + 1
ensure
        y < 3
end
```

Assignment rule:

{P (e)} x := e {P (x)}

With references (pointers)

-- True

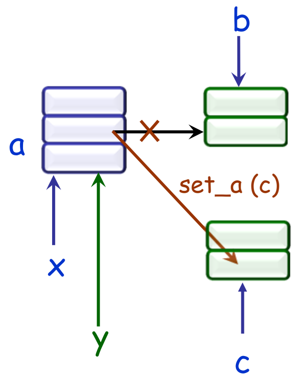
?

X.set_a (c)

-- y.a = b

-- x.a = c

Understand as
x.a := c



5

Why alias analysis is important

1. Without it, cannot apply standard proof techniques to programs involving pointers

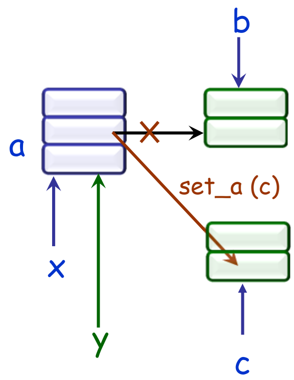
-- True

?

X.set_a (c)

-- y.a = b

-- x.a = c



2. Concurrent program analysis, in particular deadlock
3. Program optimization

6

The question under study

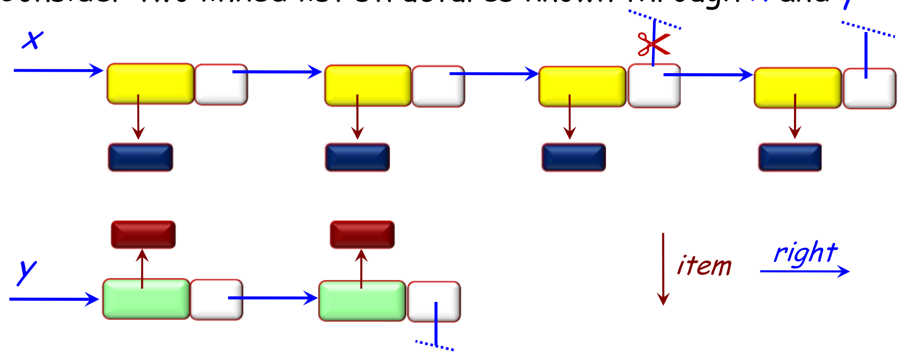
Given expressions *e* and *f* (of reference types) and a program location *p*:

At *p*, can *e* and *f* ever be attached to the same object?

(If so, we say that *e* and *f* are aliased to each other, meaning *potentially* aliased.)

An example of alias analysis

Consider two linked list structures known through x and y:



Computing the alias relation shows that:

- If x ≠ y, then no cell reachable from x (or) can be reached from y (or), and conversely
- Without this assumption, such aliasing is possible

What the calculus is about

Relation of interest:

"In the computation, e might become aliased to f"

Definition:

A binary relation is an **alias relation** if it is *symmetric* and *irreflexive*

Not necessarily transitive:

```
if c then
    x := y
else
    y := z
end
```

Can alias x to y

and y to z

but not x to z

9

What the calculus is about

The calculus defines, for any instruction p and any alias relation a, the value of

$$a \gg p$$

which denotes:

The aliasing relation resulting from executing p from an initial state in which the aliasing relation is a

For an entire program: compute $\varnothing \gg p$

10

Obtaining an alias relation

Set of binary relations on E; formally: $\mathcal{P}$(E x E)

If r is a relation in E ↔ E, the following is an alias relation:

$$\overline{r} \triangleq (r \cup r^{-1}) - Id\,[E]$$

Identity on E

Set difference

Example: $\overline{\{[\cancel{x, x}], [x, y], [y, z]\}} = \{[x, y], [y, x], [y, z], [z, y]\}$

Generalized to sets:

$$\overline{\{x, y, z\}} = \{[x, y], [y, x], [x, z], [z, x], [y, z], [z, y]\}$$

"Complete" alias relation

11

Canonical form & alias diagrams

Canonical form of an alias relation: union of complete alias relations, e.g.

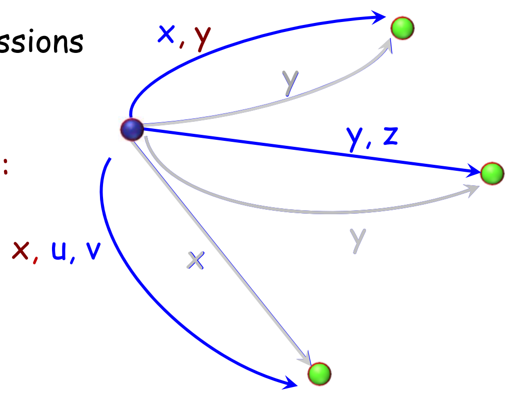
$\overline{x, y}, \overline{y, z}, \overline{x, u, v}$, meaning $\overline{\{x, y\}} \cup \overline{\{y, z\}} \cup \overline{\{x, u, v\}}$

None of the sets of expressions is a subset of another

An alias diagram:

(not canonical)

Make it canonical:



12

The alias calculus

a » **skip** = a

a » (**then** p **else** q **end**) = (a » p) ∪ (a » q)

a » (p ; q) = (a » p) » q

a » (**forget** x) = a \- {x}

a » (**create** x) = a \- {x}

a » (x := y) = a [x: y]

a » **cut** x, y = a — x, y

a » p^0 = a

a » p^{n+1} = (a » p^n) » p

a » (**loop** p **end**) = $\bigcup_{n \in \mathbb{N}}$ (a » p^n)

a » **call** r (v) = (a [x • $r^\bullet$: v]) » r

a » **call** x • r (v) = x • (x' • (**call** r (v))

a [x: y] = **given** b = a\- {x} **then** b ∪ ({x} × (b/y)) **end**

Plus:

x • **Current** = x

Current • x = x

x' • **old** x = **Current**

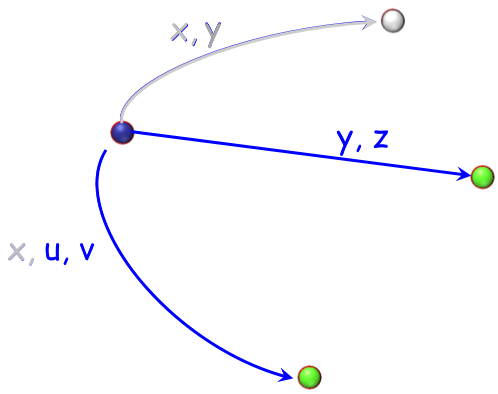
x • x' = **Current**

Current' = **Current**

The **forget** rule

a » (**forget** x) = a \- {x}

a deprived of all pairs involving x



Operations on alias relations

For an alias relation a in $E \leftrightarrow E$, an expression x, and a set of expressions $A \subseteq E$, the following are alias relations:

$$r \setminus\!- A \triangleq r - \overline{E \times A}$$

"Minus"

Set of all expressions

$$a / x \triangleq \{y : E \mid (y = x) \lor [x, y] \in a\}$$

"Quotient", similar to equivalence class in equivalence relation

15

The assignment rule (non O-O)

Value of $a \gg (x := y)$

a deprived of all pairs involving x

$$
\begin{array}{ll}
a\,[x: y] \;=\; & \textbf{given} \\
& \quad b \triangleq a \setminus\!- \{x\} \\
& \textbf{then} \\
& \quad \overline{b \cup (\{x\} \times (b / y))} \\
& \textbf{end}
\end{array}
$$

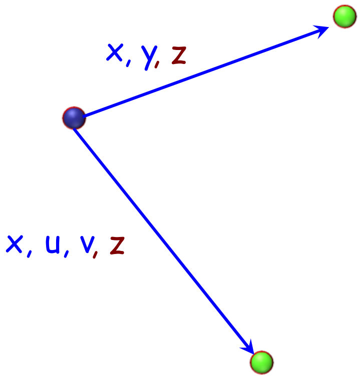
Symmetrize and de-reflect

All u aliased to y in b, plus y itself

All pairs [x, u] where u is either aliased to y in b or y itself

16

Assignment example 1

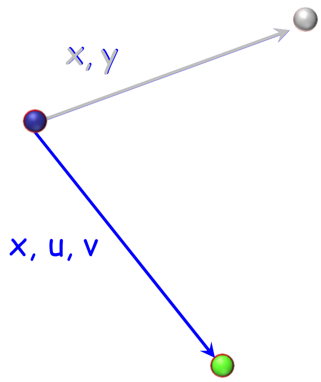


Before / After

z := x

17

Assignment example 2

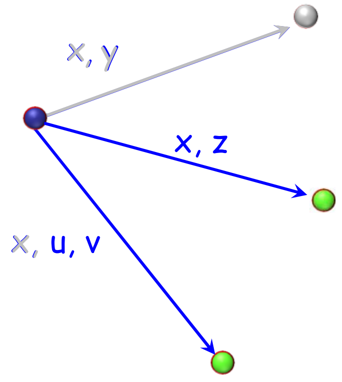


Before / After

x := u

18

Assignment example 3



Before
After

x := z

19

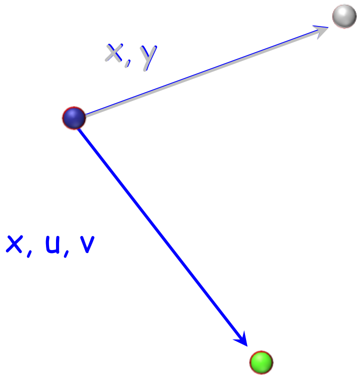
The cut instruction

cut x, y

Semantics: remove aliasing, if any, between x and y

20

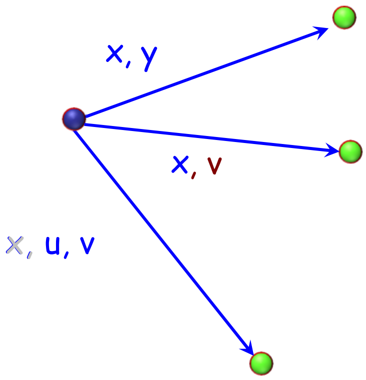
Cut example 1



Before
After

cut x, y

Cut example 2



Before
After

cut x, u

Cut rule

$a \gg$ **cut** $x, y \quad = \quad a - \overline{x, y}$

Set difference

The role of **cut**

cut x, y informs the alias calculus with non-alias properties coming from other sources

Alias relation: ∅

Example:

if m < n **then** x := u **else** x := y **end**

$\overline{x, u}, \ \overline{x, y}$

m := m + 1

if m < n **then** z := x **end**

$\overline{x, u, z}, \ \overline{x, y, z}$

But here x cannot be aliased to y (only to u). The alias theory does not know this property!

To take advantage of it, add the instruction

cut x, y;

This expression represents

check x /= y **end** (Eiffel)

assert x != y ; (JML, Spec#)

Introducing repetitions

Loop constructs:

- p^n (for integer n): n executions of p

  (auxiliary notion)

- **loop** p **end**: any sequence (incl. empty) of executions of p

Aliasing from loop constructs

$a \gg p^0 \quad = \quad a$

$a \gg p^{n+1} \quad = \quad (a \gg p^n) \gg p$ -- For $n \geq 0$

-- Also equal to $(a \gg p) \gg p^n$

$a \gg (\textbf{loop}\ p\ \textbf{end}) \quad = \bigcup_{n \in \mathbb{N}} (a \gg p^n)$

Loop aliasing theorem

$a \gg$ (**loop** p **end**) is the fixpoint of the sequence

$$t_0 = a$$
$$t_{n+1} = t_n \cup (t_n \gg p)$$

Gives a practical way to compute $a \gg$ (**loop** p **end**)

Proof: by induction. If s_n is original sequence $\bigcup_{k:\, 0..n} (a \gg p^n)$, prove separately $s_n \subseteq t_n$ and $t_n \subseteq s_n$

27

Introducing procedures

A program is now sequence of procedure definitions (one designated as main):

$r_i(f)$ **do** p_i **end**

Alias calculus notations:

- $\underline{r}$ denotes body of r (i.e. $\underline{r_i} = p_i$)
- $r^\bullet$ denotes formals of r (here f)

Instructions: as before, plus

call r_i (v) -- Procedure call

28

Handling arguments

The calculus will treat

call r (v)

i.e. $formal_1 := actual_1; \ldots ; formal_n := actual_n$

as

$r^\bullet := v$; call r

(With recursion, possible loss of precision)

Generalize notation a [x: y] to lists: use

a [u: v]

as abbreviation for

$(\ldots((a\ [u_1: v_1])[u_2: v_2])\ \ldots[u_n: v_n]$

For example: $a\ [r^\bullet: v]$

29

Call rule

a » call r (v) = $a\ [r^\bullet: v]$ » r

Formal arguments of r

Body of r

30

Using the call rule

$a \gg$ **call** $r\ (v) \quad = \quad a\ [r^{\bullet}: v] \gg \underline{r}$

Because of recursion, no longer just definition but equation

For entire set of procedures P, this gives a vector equation

$$a \gg P \quad = \quad AL\ (a \gg P)$$

Interpret as fixpoint equation and solve iteratively
(Fixpoint exists: increasing sequence on finite set)

31

Object-oriented mechanisms

Add O-O constructs:

- 1. Qualified expressions: $x \bullet y$

  Can be used as source (not target!) of assignments

  $x := y \bullet z$

- 2. Qualified calls:

  call $x \bullet r\ (v)$

- 3. ***Current***

32

Assignment (original rule)

Value of a » (x := y)

a deprived of all pairs involving x

a [x: y] = **given**

b ≜ a \- {x}

then

b ∪ ({x} × (b / y))

end

All u aliased to y in b, plus y itself

Example:

x := y•z

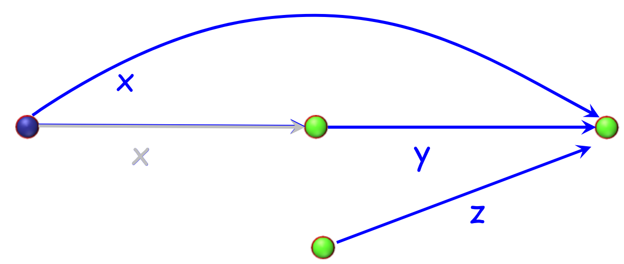
This includes [x, y] !

All pairs [x, u] where u is either aliased to y in b or y itself

33

Assigning a qualified expression

x := x•y



x does not get aliased to x•y!

(only to any z that *was* aliased to x•y)

34

Assignment rule revisited

Value of a » (x := y)

a deprived of all pairs involving x or an expression starting with x

a [x: y] = **given**

  b $\triangleq$ a \- {x}

then

  $\overline{\text{b} \cup (\{\text{x}\} \times (\text{b} / \text{y}))}$

end

Example:

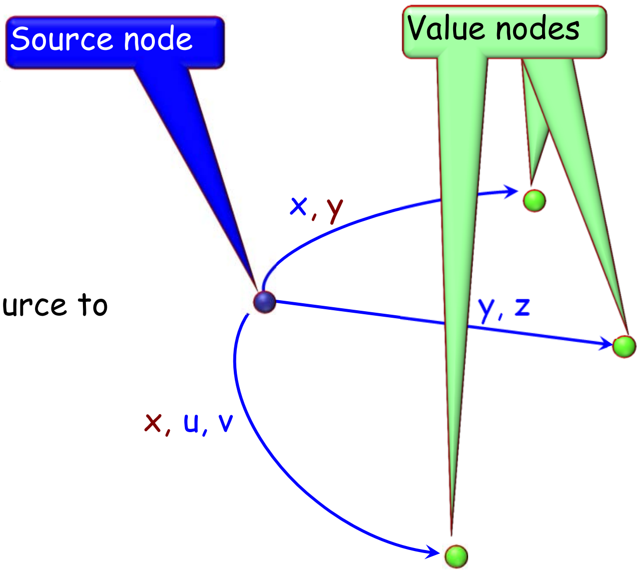
x := y • z

35

Non-O-O Alias diagrams

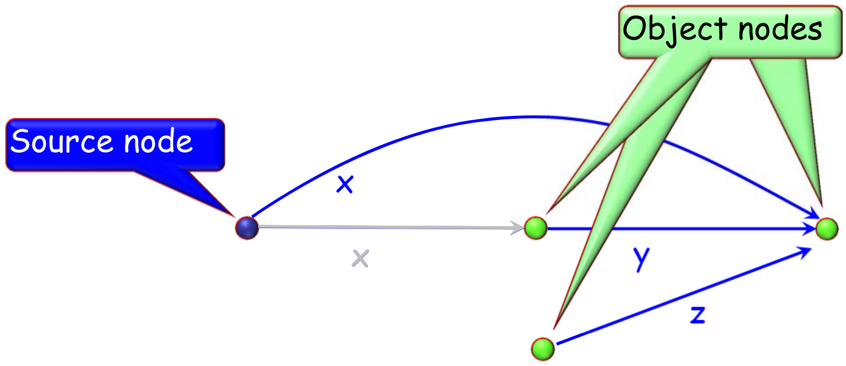
Single source node (represents stack)

Links: only from source to value



36

O-O Alias diagrams



Links may now exist between value nodes
(now called **object nodes**)

Cycles possible (see next)

Negative variables (reminder)

x • Current = x

Current • x = x

x' • old x = Current

x • x' = Current

Current' = Current

New form of call: qualified

call $x \bullet r \; (a, b, \ldots)$

Distribution operator (reminder): ▪

For a list $v = \langle u, v, w, \ldots \rangle$:

$$x \blacksquare v \quad = \langle x \bullet u, x \bullet v, x \bullet v, \ldots \rangle$$

For a relation r in $E \leftrightarrow E$:

$$x \blacksquare r \quad = \{[x \bullet a, x \bullet b] \mid [a, b] \in r\}$$

Example:

$$x \blacksquare (\overline{u, v, w}, \; \overline{u, y}) \quad = \quad \overline{x \bullet u, x \bullet v, x \bullet w}, \; \overline{x \bullet u, x \bullet y}$$

The qualified call rule

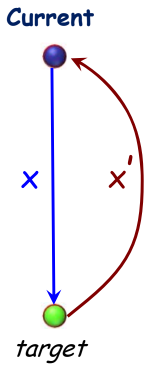
a » **call** x•r (v) = x ▪ (x' ▪ (**call** r (v))

Treat

call x•r (v)

as

x▪formals := v ; **call** x•r



Processing a qualified call

a » **call** x•r = x ▪ ((x' ▪ a) » <u>r</u>)

Alias relation:

$\overline{\text{c, d}}$

Prefix with x' ▪ :

$\overline{\text{x'•c, x'•d}}$

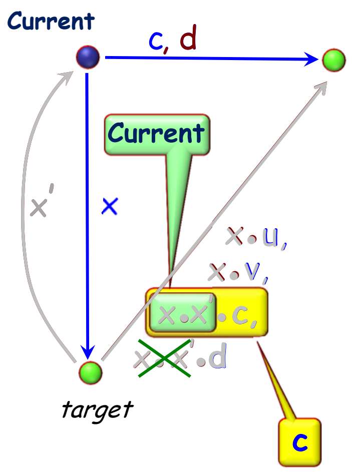
$\overline{\text{u, x'•c, x'•d}}$

$\overline{\text{v, u, x'•c, x'•d}}$

Prefix with x ▪ :

$\overline{\text{x•v, x•u, c, d}}$

```
d := c
call x•r
with
r
  do
    u := x'•c
    v := u
  end
```



Seen from the remote side

a » **call** x•r = x ▪ ((x' ▪ a) » <u>r</u>)

Alias relation:

$\overline{\text{c, d}}$

Prefix with x' ▪ :

$\overline{\text{x'•c, x'•d}}$

$\overline{\text{u, x'•c, x'•d}}$

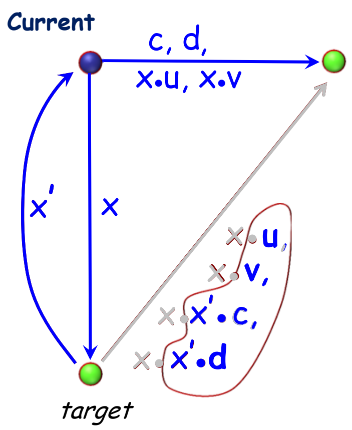
$\overline{\text{v, u, x'•c, x'•d}}$

Prefix with x ▪ :

$\overline{\text{x•v, x•u, c, d}}$

E4 version:

```
d := c
call x•r
with
r
  do
    u := x'•c
    v := u
  end
```



43

Termination?

The original termination argument does not hold any more

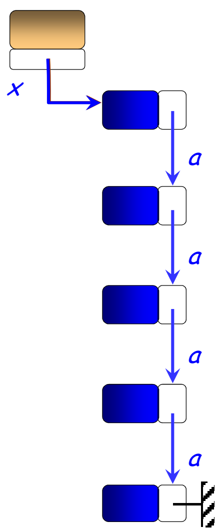
Consider

from y := x **loop**
    y := y•a
end

y may become aliased to:

x, x•a, x•a•a, x•a•a•a etc.

(infinite set of expressions!)



44

Termination: the question under study

Given expressions *e* and *f* (of reference types) and a program location *p*:

At *p*, can *e* and *f* ever be attached to the same object?

The alias calculus

a » **skip**	=	a
a » (**then** p **else** q **end**)	=	(a » p) ∪ (a » q)
a » (p ; q)	=	(a » p) » q
a » (**forget** x)	=	a \- {x}
a » (**create** x)	=	a \- {x}
a » (x := y)	=	a [x: y]
a » **cut** x, y	=	a — x, y
a » p^0	=	a
a » p^{n+1}	=	(a » p^n) » p
a » (**loop** p **end**)	=	$\bigcup_{n \in \mathbb{N}}$ (a » p^n)
a » **call** r (v)	=	(a [x ▪ r•: v]) » <u>r</u>
a » **call** x•r (v)	=	x ▪ ((x' ▪ a) » (**call** r (x' ▪ v))

a [x: y] = **given** b = a\- {x} **then**
b ∪ ({x} × (b/y))
end

Plus:

x•**Current**	=	x
Current•x	=	x
x'• **old** x	=	**Current**
x•x'	=	**Current**
Current'	=	**Current**

Approaches for comparison

Separation logic

Shape analysis

Ownership

Dynamic frames

Achievements

Theory of aliasing
Simple (about a dozen rules)
New concepts: inverse variables, modeling **Current**
Graphical formalism (alias diagrams), canonical form
Implemented
Almost entirely automatic (except for occasional **cut**)
Small loss of precision, i.e. not too conservative
Abstract: does not mention stack and heap
Covers object-oriented programming
Faithful to O-O spirit; see qualified call rule

$$\boxed{a \gg \textbf{call}\ x \bullet f \quad = \quad x \cdot ((x' \cdot a) \gg \textbf{call}\ f)}$$

Can cover full modern O-O language
Potential solution to "frame problem"

Limitations and future work

Extend for polymorphism and dynamic binding

Use a more modular approach

Apply to solving frame problem

Integrate with standard axiomatic reasoning

Integrate implementation with compiler