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)\}

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

Assignment rule:
\{P (e)\} x := e \{P (x)\}
With references (pointers)

--- True

\textbf{X.set\_a (c)}

\begin{itemize}
  \item \texttt{y.a = b}
  \item \texttt{x.a = c}
\end{itemize}

\textbf{Understand as} \texttt{x.a := c}

Why alias analysis is important

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

--- True

\textbf{X.set\_a (c)}

\begin{itemize}
  \item \texttt{y.a = b}
  \item \texttt{x.a = c}
\end{itemize}

2. Concurrent program analysis, in particular deadlock

3. Program optimization
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 $x \neq y$, then no cell reachable from $x$ (yellow or red) can be reached from $y$ (green or blue), 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:

\[
\begin{align*}
\text{if } & c \text{ then } \\
& x := y \\
\text{else } & y := z \\
& y := z \\
\text{end}
\end{align*}
\]

Can alias \(x\) to \(y\)
and \(y\) to \(z\)
but not \(x\) to \(z\)

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

\[a \rightarrow p\]

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 \(\emptyset \rightarrow p\)
Obtaining an alias relation

If \( r \) is a relation in \( E \leftrightarrow E \), the following is an alias relation:

\[ r \mapsto (r \cup r^{-1}) - \text{Id} [E] \]

Example: \( \{[x, x], [x, y], [y, z]\} = \{[x, y], [y, x], [y, z], [z, y]\} \)

Generalized to sets:

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

Canonical form & alias diagrams

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

\[ \{x, y, y, z, x, u, v\} \], meaning \( \{x, y\} \cup \{y, z\} \cup \{x, u, v\} \)

None of the sets of expressions is a subset of another

An alias diagram: (not canonical)

Make it canonical:
### The alias calculus

<table>
<thead>
<tr>
<th>Rule</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a » skip</code></td>
<td><code>a</code></td>
</tr>
<tr>
<td><code>a » (then p else q end)</code></td>
<td><code>(a » p) ∪ (a » q)</code></td>
</tr>
<tr>
<td><code>a » (p : q)</code></td>
<td><code>(a » p) » q</code></td>
</tr>
<tr>
<td><code>a » (forget x)</code></td>
<td><code>a \- {x}</code></td>
</tr>
<tr>
<td><code>a » (create x)</code></td>
<td><code>a \- {x}</code></td>
</tr>
<tr>
<td><code>a » (x := y)</code></td>
<td><code>a [x: y]</code></td>
</tr>
<tr>
<td><code>a » cut x, y</code></td>
<td><code>a \- x, y</code></td>
</tr>
<tr>
<td><code>a » p^0</code></td>
<td><code>a</code></td>
</tr>
<tr>
<td><code>a » p^{n+1}</code></td>
<td><code>(a » p^n) » p</code></td>
</tr>
<tr>
<td><code>a » (loop p end)</code></td>
<td><code>\bigcup_{n \in \mathbb{N}} (a » p^n)</code></td>
</tr>
<tr>
<td><code>a » call r(v)</code></td>
<td><code>(a [x \* r^* : v]) » r</code></td>
</tr>
<tr>
<td><code>a » call x^* r(v)</code></td>
<td><code>x^* (x^* (call r(v)))</code></td>
</tr>
</tbody>
</table>

### 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:

\[
\begin{align*}
    r \setminus A & \triangleq r - E \times A \\
    a / x & \triangleq \{y \in E \mid (y = x) \lor [x, y] \in a\}
\end{align*}
\]

"Quotient", similar to equivalence class in equivalence relation

The assignment rule (non O-O)

Value of \( a \rightarrow (x := y) \)

\[
\begin{align*}
    a [x: y] & \quad \text{given} \\
    b & \triangleq a \setminus \{x\} \\
    \text{then} \\
    b & \cup ((x) \times (b / y))) \\
    \text{end}
\end{align*}
\]

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
Assignment example 1

Before

\[ z := x \]

After

\[ x, y, z \]

\[ x, u, v, z \]

Assignment example 2

Before

\[ x := u \]

After

\[ x, y \]

\[ x, u, v \]
Assignment example 3

Before

\[ x := z \]

The cut instruction

\[ \text{cut } x, y \]

Semantics: remove aliasing, if any, between \( x \) and \( y \)
Cut example 1

Before

![Diagram showing cut example 1]

After

Cut example 2

Before

![Diagram showing cut example 2]

After
The role of **cut**

**cut** $x, y$ informs the alias calculus with non-alias properties coming from other sources. Example:

```
if m < n then x := u else x := y end
m := m + 1
if m < n then z := x end
```

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

- $a \triangleright p^0 = a$
- $a \triangleright p^{n+1} = (a \triangleright p^n) \triangleright p$  -- For $n \geq 0$
  -- Also equal to $(a \triangleright p) \triangleright p^n$
- $a \triangleright (\text{loop } p \text{ end}) = \bigcup_{n \in \mathbb{N}} (a \triangleright p^n)$
Loop aliasing theorem

\[ a \rightarrow \text{(loop } p \text{ end)} \text{ is the fixpoint of the sequence} \]
\[
\begin{align*}
t_0 & = a \\
t_{n+1} & = t_n \cup (t_n \rightarrow p)
\end{align*}
\]

Gives a practical way to compute \( a \rightarrow \text{(loop } p \text{ end)} \)

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

Introducing procedures

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

\[
\begin{align*}
\text{r}_i(f) & \text{ do } p_i \text{ end} \\
\end{align*}
\]

Alias calculus notations:
\[ r \text{ denotes body of } r \text{ (i.e. } r_i = p_i) \]
\[ r^* \text{ denotes formals of } r \text{ (here } f) \]

Instructions: as before, plus

\[
\begin{align*}
\text{call } r_i(v) \\
\end{align*}
\]

-- Procedure call
Handling arguments

The calculus will treat
\[ \text{call } r(v) \]
as
\[ r^* := v ; \quad \text{call } r \]
(With recursion, possible loss of precision)

Generalize notation \( a [x; y] \) to lists: use \( a [u; v] \) as abbreviation for
\[ \cdots ((a [u_1; v_1])[u_2; v_2]) \cdots [u_n; v_n] \]
For example: \( a [r^*; v] \)

Call rule

\[ a \Rightarrow \text{call } r(v) = a [r^*; v] \Rightarrow r \]

Formal arguments of \( r \)
Body of \( r \)
Using the call rule

\[ a \Rightarrow \text{call } r(v) = a[r^*;v] \Rightarrow r \]

Because of recursion, no longer just definition but equation

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

\[ a \Rightarrow P = AL(a \Rightarrow P) \]

Interpret as fixpoint equation and solve iteratively

(Fixpoint exists: increasing sequence on finite set)

Object-oriented mechanisms

Add O-O constructs:

- 1. Qualified expressions: \( x.y \)

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

  \[ x := y.z \]

- 2. Qualified calls:

  \[ \text{call } x.r(v) \]

- 3. Current
Assignment (original rule)

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

$\begin{align*}
\text{a \hspace{1em} given} \\
\text{b \triangleleft a \setminus \{x\}} \\
\text{then} \\
\text{b \cup \{(x) \times (b / y)\}} \\
\text{end}
\end{align*}$

Example:

$x := y \cdot z$

This includes $[x, y]$!

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

All $u$ aliased to $y$ in $b$, plus $y$ itself

Assigning a qualified expression

$x := x \cdot y$

$x$ does not get aliased to $x \cdot y$!

(only to any $z$ that was aliased to $x \cdot y$)
Assignment rule revisited

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

$$a [x: y] = \begin{array}{l}
\text{given} \\
b \triangleq a \setminus \{x\}
\end{array}$$

then

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

end

Example:

$x := y \cdot z$

Non-O-O Alias diagrams

Single source node (represents stack)

Links: only from source to value

Source node

Value nodes
O-O Alias diagrams

Links may now exist between value nodes (now called object nodes)
Cycles possible (see next)

Negative variables (reminder)

\[
\begin{align*}
x \cdot \text{Current} &= x \\
\text{Current} \cdot x &= x \\
x' \cdot \text{old} \cdot x &= \text{Current} \\
x \cdot x' &= \text{Current} \\
\text{Current}' &= \text{Current}
\end{align*}
\]
New form of call: qualified

call $x \cdot r (a, b, ...)$

Distribution operator (reminder):

For a list $v = <u, v, w, ...>$:

$$x \cdot v = <x \cdot u, x \cdot v, x \cdot v, ...>$$

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

$$x \cdot r = \{ [x \cdot a, x \cdot b] \mid [a, b] \in r \}$$

Example:

$$x \cdot (u, v, w, u, y) = x \cdot u, x \cdot v, x \cdot w, x \cdot u, x \cdot y$$
The qualified call rule

\[ a \rightarrow \text{call } x \cdot r(v) = x' \cdot (x' \cdot (\text{call } r(v))) \]

Treat

\[ \text{call } x \cdot r(v) \]

as

\[ x \cdot \text{formals} := v ; \text{call } x \cdot r \]

Processing a qualified call

\[ a \rightarrow \text{call } x \cdot r = x' \cdot (x' \cdot a) \rightarrow r \]

Alias relation:

\[ c, d \]

Prefix with \( x' \cdot \):

\[ x' \cdot c, x' \cdot d \]

Prefix with \( x' \cdot c, x' \cdot d \)

\[ u, x' \cdot c, x' \cdot d \]

Prefix with \( x' \cdot c, x' \cdot d \)

\[ v, u, x' \cdot c, x' \cdot d \]

Prefix with \( x \cdot \):

\[ x \cdot v, x \cdot u, c, d \]
**Seen from the remote side**

\[ a \rightarrow \text{call } x \cdot r = x \cdot ((x' \cdot a) \rightarrow r) \]

**Alias relation:**

- \[ c, d \]

**Prefix with \( x' \):**

- \[ x' \cdot c, x' \cdot d \]
- \[ u, x' \cdot c, x' \cdot d \]
- \[ v, u, x' \cdot c, x' \cdot d \]

**Prefix with \( x \):**

- \[ x \cdot v, x \cdot u, c, d \]

**E4 version:**

\[ d := c \]
\[ \text{call } x \cdot r \]
\[ \text{with } r \]
\[ \text{do} \]
\[ u := x' \cdot c \]
\[ v := u \]
\[ \text{end} \]

**Current**

\[ c, d, x \cdot u, x \cdot v \]

**Target**

\[ x' \]

**Termination?**

The original termination argument does not hold any more

Consider

\[ \text{from } y := x \text{ loop} \]
\[ y := y \cdot a \]
\[ \text{end} \]

\( y \) may become aliased to:

- \( x, x \cdot a, x \cdot a \cdot a, x \cdot a \cdot a \cdot a \) etc.

(infinite set of expressions!)
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

<table>
<thead>
<tr>
<th>Rule</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a \gg \text{skip}$</td>
<td>$a$</td>
</tr>
<tr>
<td>$a \gg (\text{then } p \text{ else } q \text{ end})$</td>
<td>$(a \gg p) \cup (a \gg q)$</td>
</tr>
<tr>
<td>$a \gg (p ; q)$</td>
<td>$(a \gg p) \gg q$</td>
</tr>
<tr>
<td>$a \gg (\text{forget } x)$</td>
<td>$a \setminus {x}$</td>
</tr>
<tr>
<td>$a \gg (\text{create } x)$</td>
<td>$a \setminus {x}$</td>
</tr>
<tr>
<td>$a \gg (x := y)$</td>
<td>$a \setminus {x: y}$</td>
</tr>
<tr>
<td>$a \gg \text{cut } x, y$</td>
<td>$a - x, y$</td>
</tr>
<tr>
<td>$a \gg p^0$</td>
<td>$a$</td>
</tr>
<tr>
<td>$a \gg p^{n+1}$</td>
<td>$(a \gg p^n) \gg p$</td>
</tr>
<tr>
<td>$a \gg (\text{loop } p \text{ end})$</td>
<td>$\bigcup_{n \in \mathbb{N}} (a \gg p^n)$</td>
</tr>
<tr>
<td>$a \gg \text{call } r (v)$</td>
<td>$(a \gg [x: r^*: v]) \gg r$</td>
</tr>
<tr>
<td>$a \gg \text{call } x \cdot r (v)$</td>
<td>$x \cdot ((x' \cdot a) \gg (\text{call } r (x' \cdot v)))$</td>
</tr>
</tbody>
</table>

**Plus:**

- $x \cdot \text{Current} = x$
- $\text{Current} \cdot x = x$
- $x \cdot \text{old } x = \text{Current}$
- $x \cdot x' = \text{Current}$
- $\text{Current}' = \text{Current}$
**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

\[
a \rightarrow \text{call} \ x. f \quad = \quad x \cdot ((x' \cdot a) \rightarrow \text{call} f)
\]

- 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