
The theory and calculus of aliasing

Bertrand Meyer

se.ethz.ch

Abstract. A theory, graphical notation, mathematical calculus and implementation for finding whether two given expressions can, at execution time, denote references attached to the same object. Intended as the basis for a comprehensive solution to the “frame problem” and as an alternative (for the specific issue of determining aliases) to separation logic, shape analysis, ownership types and dynamic frames.

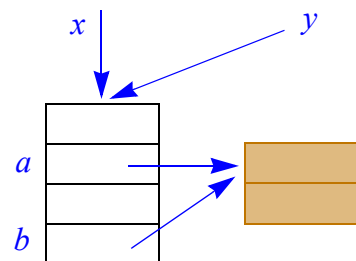
1 Dynamic aliasing

You have, most certainly, read Homer. I have not (too much blood), but then I listen to Offenbach a lot, so we share some knowledge: we both understand that “*the beautiful daughter of Leda and the swan*”, “*poor Menelaus’s spouse*” and “*Pâris’s lover*” all denote the same person, also known as *Helen of Troy*. The many modes of calling Helen are a case of *aliasing*, the human ability to denote a single object by more than one name.

Aliasing is at its highest risk of causing confusion when it is *dynamic*, that is to say when an object can at any moment acquire a new name, especially if that name previously denoted another object. The statement “*I found Pâris’s lover poorly dressed*” does not necessarily cast aspersion on Helen’s sartorial tastes, as Pâris might by now have found himself a new lover; but if we do not carefully follow the lives of the rich and famous we might believe it does.

Stories of dire consequences of dynamic aliasing abound in life, literature and drama. There is even an opera, Smetana’s *The Bride Sold*¹, whose plot *entirely* rests on a single aliasing event. To the villagers’ dismay, Jeník promises the marriage broker, in return for good money, not to dissuade his sweetheart Mařenka from marrying the son of the farmer Mícha. Indeed Mícha wants Mařenka for his dimwit son, Vašek, but it is suddenly revealed that Jeník, believed until then to be a stranger to the village, is Mícha’s son from a first marriage: he has tricked everyone.

To a programmer, this tale sounds familiar: the equivalent in program execution is to perform an operation on certain operands, and inadvertently to modify a property of a target that is not named in the operation — hence the risk of confusion — but aliased to one of the operands. For example an operation may, officially, modify the value of $x.a$; but if x denotes a reference and y another reference which happens at the time of execution to be aliased to x (meaning that they both point to the same object), the operation will have an effect on $y.a$ even though its text does not cite y . If b is aliased to a , we might even have an operation that modifies $y.b$ although its description in the programming language mentions neither y nor b .



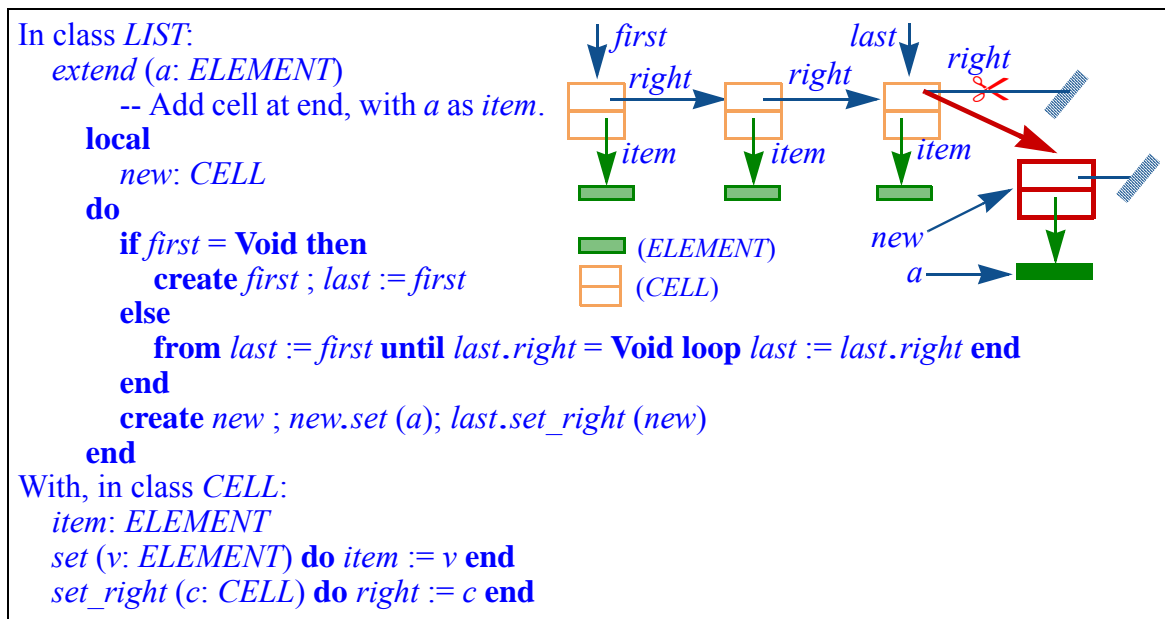
1. A title incorrectly rendered, in the standard English translation, as *The Bartered Bride*.

It is not hard to justify the continued search for effective verification techniques covering aliasing. In the current state of proof technology, the aliasing problem (together with the associated *frame* problem, to which it provides the key) is the principal obstacle on the road to full proofs of correctness for sequential programs; it also plays a role in the specific difficulties of proving *concurrent* programs correct.

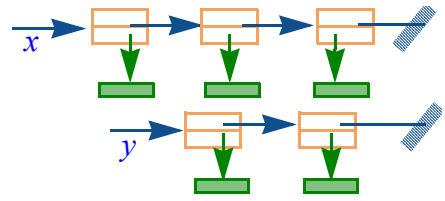
A symptom of this situation is that industrial-grade proving tools often preclude the use of pointers altogether. The Spark environment, which has made a remarkable contribution towards showing that production programs can be routinely subjected to proof requirements, provides a striking example. Spark relies on a programming language, presented as a subset of Ada but in reality a subset of a Pascal-like language (plus modules), without support for pointers or references. In considering how to make such pioneering advances relevant to a larger part of the industry, it is hard to imagine masses of programmers renouncing pointers and other programming languages advances of the past three decades.

The absence of a generally accepted solution is not due to lack of trying. The aliasing problem has been extensively researched, and interesting solutions proposed, in particular *shape analysis*, *separation logic*, *ownership types* and *dynamic frames*. Few widely used proof environments have integrated these techniques. That may still happen, but the obstacles are significant; in particular, the first two approaches suffer (in our opinion) by attempting to draw a picture of the run-time pointer structure that is more precise than needed for alias analysis; and the last three assume a supplementary annotation effort (in addition to standard Hoare assertions) at which programmers may balk.

The theory, calculus and prototype implementation described in the present work strive to avoid these limitations. An example of a typical problem that they directly address is the absence of any aliasing between any elements of two linked lists created and modified through typical object-oriented techniques. Assume a standard implementation of lists with an operation to add elements at the end:



Consider references x and y denoting two such lists built through any number of applications of *extend* and similar operations. The theory, and its implementation presented below, determine that if they are not aliased to each other ($x \neq y$) no *CELL* or *ELEMENT* reachable from x is also reachable from y . The proof is entirely automatic: it does not require any annotation. In the implementation, it is instantaneous.



In its present state the theory suffers from some limitations (section 9), but it makes the following claims:

- It provides a comprehensive treatment of aliasing issues and, potentially, a solution to the “frame problem”.
- It includes a graphical notation, *alias diagrams*, which helps reason about aliasing.
- Alias analysis is almost entirely automatic, requiring no assertions or other annotations from the programmer. The only exception is the occasional need to add a **cut** instruction (4.4) to inform the calculus with results obtained from other sources; this case should arise only rarely. Outside of it, alias analysis enjoys the advantage often invoked in favor of model checking and abstract interpretation against annotation-based approaches to program proving: full automation.
- The loss of precision (inevitable because of the undecidability of aliasing in its general form) is usually acceptable, and, when not, can be addressed through **cut**.
- The theory is at a suitably high level of abstraction, avoiding explicit references to such implementation-oriented concepts as “stack” and “heap”.
- The theory can model the full extent of a modern object-oriented language.
- The reader will, it is hoped, agree that it is simple (about a dozen rules) and provides insights into the nature of programming, especially object-oriented programming. An example is the final rule /36/, for qualified calls: $(a \mid= \text{call } x.r) = (x \bullet ((x' \bullet a) \mid= \text{call } r))$, which conveys the essence of the fundamental mechanism of O-O computation, concisely capturing the notion of current object and the principle of relativity, both central to the O-O model.

The following ideas are believed to be new (although of course heavily influenced by previous work): the notion of alias calculus; alias diagrams (a simplification of “shape graphs”); the canonical form of alias relations; limiting analysis to expressions occurring in the program; using alias analysis as a preprocessing step for axiomatic-style proofs; **cut**; inverted variables; the handling of arguments, loops and conditionals.

The ambition behind the present work is that it will complement the methods listed earlier and, for the problem of determining aliases (which is only a part of their scope), possibly provide an alternative.

Section 2 sets the context. Section 3 describes the properties of alias relations. Section 4 introduces the calculus for a simple language without remote object access, which section 5 extends with procedures. Section 6 generalizes the language and the calculus to the target domain of interest: object-oriented programming. Section 7 presents the prototype implementation. Section 8 summarizes how to apply the calculus to an actual object-oriented programming language. Section 9 lists the remaining problems.

All the examples of this article can be tried out in the implementation, which the reader can download (as a Windows executable) from se.ethz.ch/~meyer/down/alias.zip.

2 General observations

The goal of the calculus is to allow deciding whether two reference expressions appearing in a program might, during some execution, have the same value, meaning that the associated references are attached to the same object.

2.1 Adding the alias calculus to an axiomatic framework

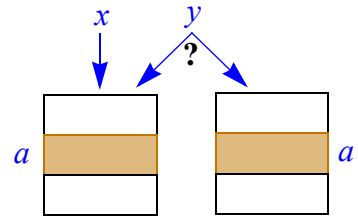
The key to the simplicity of the calculus is the expectation that cases of aliasing are, in practice, the exception: most of the time, two expressions are *not* aliased to each other. As a consequence, the intended approach to program proving is an incremental modification of standard axiomatic (Hoare-style) techniques:

- 1 • A first step uses the alias calculus to determine the possible aliases of expressions that appear in assertions.
- 2 • The second step applies standard axiomatic reasoning to the program equipped with the resulting set of assertions — the original enriched with alias variants.

The techniques used in these two steps are independent. Step 2 uses ordinary axiomatic semantics (including backward reasoning because of the assignment axiom); step 1 uses the calculus (which happens to work in a forward style).

The following example illustrates the process. Assume we are asked to prove

<pre>{not y.a} x.set_a {not y.a}</pre>	/1/
--	-----



We are dealing with objects having a boolean attribute a , which the procedure `set_a` sets to **True**. Assume that we have at our disposal a proof framework (not detailed here, but relying on standard techniques) that applies axiomatic semantics, enabling us to prove

<pre>{True} x.set_a {x.a}</pre>	/2/
---------------------------------	-----

The proof of /2/ will involve the assignment axiom, as `set_a` performs $a := \mathbf{True}$, and a procedure rule. (If we informally understand the call as $x.a := \mathbf{True}$, the proof is a trivial application of the assignment axiom.)

If we naïvely applied similar techniques to prove /1/, the proof would proceed smoothly: since the instruction does not name y , the postcondition sails through that instruction unchanged. Such reasoning, however, is not sound if y can be aliased to x . The alias calculus will allow us, through its own techniques distinct from axiomatic semantics, to determine possible aliasings. If it finds that some computations might alias y to x , it will inform the axiomatic reasoning by automatically enriching the postcondition of /1/ to read **not $y.a$ and not $x.a$** . Then /1/ is no longer a correct Hoare triple since application of the assignment axiom to **not $x.a$** yields the weakest precondition **False**.

2.2 Handling imprecision

The theory and calculus will be defined in terms of successive programming languages of increasing ambition, each a superset of the previous one: E0 introduces variables and basic instructions; E1 introduces procedures; E2 introduces object-oriented mechanisms. To apply the calculus in practice, it will be necessary to translate the programming language of interest (such as a modern O-O language) into E2. Elements of the following discussion, and the summary in section 8, describe how to perform the translation and, as a consequence, how to apply the calculus to practical programs.

Until then, we will concentrate on the calculus itself. We must, however, note the principal property of the translation: it must be **sound**, meaning that if two expressions in the original language may become aliased in some execution the calculus must reflect that property. In the reverse direction, there is no such exigency: the calculus might infer possible aliasing between two translated expressions where no aliasing can occur between the original expressions, a case we may call *imprecision*. We will, as we go, keep on the alert for cases where the translation may introduce imprecision.

Imprecision is an inevitable risk of any practical approach to alias analysis, but might prevent some program proofs because of the possible loss of information. The alias theory introduces a special solution to this problem in the form of the **cut** instruction (section 4.4). A **cut** corrects any undesired imprecision resulting from the simplifications of the alias calculus by stating that two expressions are *not* aliased at a particular point of the program. The alias calculus itself is not, in such cases, able to prove this property; the proof falls back on its partner in the proof duo — axiomatic semantics. As an example, consider

if not <i>cond</i> then		
<i>x</i> := <i>y</i>		/3/
end		
<i>Other_instructions</i>	-- Not affecting any of <i>cond</i> , <i>x</i> and <i>y</i> .	
if <i>cond</i> then		/4/
<i>z</i> := <i>x</i>		/5/
end		

The alias theory correctly determines that at the start of the conditional instruction /4/ *x* may be aliased to *y* as a result of the earlier conditional assignment /3/. It will also determine, as a consequence, that the assignment /5/ may alias *z*, through *x*, to *y*. Such aliasing cannot occur in practice because of the role of the condition *cond*. The alias calculus, however, has no way of establishing that no run-time execution path can include both /3/ and /5/; such a property is beyond its scope. If the imprecision is unacceptable — in other words, if the spurious aliasing of *z* to *x* precludes proving the properties of interest — the prover must add a **cut** instruction to the second conditional, which becomes

if <i>cond</i> then
cut <i>x</i> , <i>y</i>
<i>z</i> := <i>x</i>
end

For the alias calculus, the **cut** instruction is a guarantee from the environment (as provided by **require** in Eiffel and **assume** in JML and Spec#) that $x \neq y$. For the axiomatic proof framework, it is a proof obligation (**check** in Eiffel, **assert** in JML and Spec#).

2.3 Scope of the theory

The purpose of the alias theory and calculus is to answer a specific question:

The aliasing question

Given two expressions of a program, e and f , of reference type, and a program point p , can e and f ever be attached to the same object during an execution of the program?

In line with the preceding observations, the calculus looks for a sound but possibly imprecise answer: it *may* — as rarely as possible — answer “yes” even if e and f could never become aliased in actual executions; but if they can, the calculus is *required* to answer “yes”.

The most important word of the above definition is the first one: “*Given*”. What makes the calculus possible is that it takes the pragmatic view of an existing program, possibly equipped with assertions. Then program proofs do not need to know all aliasing properties of all possible expressions; they only need the properties of expressions actually appearing in the program and its assertions. Expressions not named in the program are no more interesting to the prover than (except to the philosopher) the tree that falls unheard and unseen in the forest.

This observation allows us, in addition, to consider finite sets only. Without it, the analysis of a typical data structure traversal loop such as

```

from
   $x := first$ 
until some_condition loop
   $x := x.right$ 
end

```

would have to reflect that x can become aliased to $first$, $first.right$, $first.right.right$ and so on, an infinite list of expressions. It might even lead us to extend the assertion language with a regular-expression-like notation (such as $first.right^*$) to cover the possible values. While the alias calculus could be extended to handle such extensions, it does not need them for the fundamental applications discussed here.

3 Alias relations

The theory relies on a notion of “alias relation”, describing the possible aliasings between variables and expressions of a program.

3.1 Definition

Definition: alias relation

A relation in $E \leftrightarrow E$ for some set E is an alias relation if it is symmetric and irreflexive.

$E \leftrightarrow E$, defined as $\mathbb{P}(E \times E)$, is the set of binary relations on E . For our needs E will be a set of variables and expressions in a program. The presence of a pair $[x, y]$ in an alias relation associated with a program point expresses that x and y may be attached to the same object at that program point during some execution.

Such a relation must be symmetric. As to irreflexivity, we might take the reverse convention (reflexivity on E), considering that every expression is aliased to itself; such trivial aliasing obscures the interesting cases, however, and choosing irreflexivity yields simpler rules.

If $r1$ and $r2$ are alias relations, so are $r1 \cup r2$ and $r1 \cap r2$.

If r is relation, but not necessarily an alias relation, \bar{r} will denote the alias relation obtained from r by removing all reflexive pairs and symmetrizing all pairs; for example $\overline{\{[x, x], [x, y], [y, z]\}}$ is the alias relation $\{[x, y], [y, x], [y, z], [z, y]\}$. Formally, \bar{r} is $(r \cup r^{-1}) - Id[E]$ where “ $-$ ” is set difference and $Id[E]$ is the identity relation on E . If r is an alias relation, then $\bar{r} = r$. It is useful to extend the notation to a subset A of E , defining \bar{A} as $\overline{A \times A}$. ($A \times A$ is the “universal” relation involving all pairs in A .) So $\overline{\{x, y, z\}}$ is $\{[x, y], [y, x], [x, z], [z, x], [y, z], [z, y]\}$.

For a set A described by extension, as in this example, we may omit the braces, writing just $\overline{x, y, z}$. We may express any alias relation in a union form $\overline{T, U, V, \dots}$, meaning $\overline{T \cup U \cup V \dots}$, where every operand is a universal relation with reflexive pairs removed. With this notation, we may write the first example, $\{[x, y], [y, x], [y, z], [z, y]\}$, as $\overline{x, y, y, z}$.

An alias relation need not be transitive, as illustrated by the program extract

```

if cond then
     $x := y$ 
else
     $x := z$ 
     $u := x$ 
end

```

which (as the alias calculus will determine) yields the alias relation $\overline{x, y, x, u, z}$ but does not cause aliasing between y and z .

3.2 Canonical form and alias diagrams

An alias relation may have several union forms; for example the union forms $\overline{x, y}$, $\overline{x, u, z}$ and $\overline{x, y, x, u}$, $\overline{x, u, x, u, z}$ denote the same relation. The first of these variants, like the other examples given previously, is a *canonical form*:

Canonical form of an alias relation

The canonical form of an alias relation a is a union form $\overline{T, U, V, \dots}$ where:

- None of the sets T, U, V, \dots is a subset (proper or improper) of any of the others.
- Adding or removing any element to or from any of them would invalidate the property $\overline{T \cup U \cup V \dots} = a$.

Canonical form theorem: For any alias relation a , the canonical form exists and is unique.

Proof: consider all subsets of E . Retain only those whose elements are all aliased to each other in a . Then remove any that is a subset of another. The resulting subset of $\mathbb{P}(E)$ gives the canonical form.

Although this is a constructive proof, an algorithm applying it directly to display the canonical form of a relation would be exponential in the size of E ; the implementation uses a more efficient algorithm.

Corollary: each one of the sets T, U, V, \dots involved in a canonical form has at least two elements (since an alias reflection is irreflexive).

The reverse theorem also holds: a canonical form defines an alias relation uniquely. All alias relations for the examples that follow will be given in canonical form.

Alias diagrams are useful to visualize the theory and in particular the canonical form theorem. An alias diagram is a labeled directed graph with one special *source node* representing a program point and any number of *value nodes* each representing a set of possible values (not explicitly specified) in associated program states. At this stage of the theory, the graph is acyclic, the start node of any edge is the source node, and the end node is a value node; when we extend the theory to object-oriented programming in section 6, there will also be edges connecting value nodes.

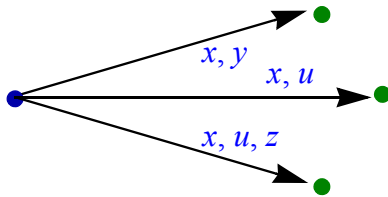
An edge is labeled by a non-empty set of expressions, for example e, f . The presence of an expression e in an edge leading to a value node n expresses that e may at the given program point have one of the values associated with n . An example alias diagram is:



The alias relation associated with such a graph is simply the set of pairs $[e, f]$ such that e and f both appear in the label for some edge. So the above graph represents the earlier example $\overline{x, y, x, u, z}$.

A value node carries no information other than its existence and the label of the edge (a single one at this stage of the theory) that leads to it. In the following discussion, as a consequence, “removing an edge” also implies removing the target node.

A diagram is in canonical form if no label is a subset of another. The canonical form theorem is easy to visualize on alias diagrams: a non-canonical diagram such as



represents the same alias relation as the previous one, so the edge labeled x, u is useless. To turn an arbitrary diagram into canonical form, remove any edge whose label is a subset of another edge’s label (and, per the general convention, remove the edge’s end node).

As a consequence of the corollary of the canonical form theorem, the label of every edge includes at least two expressions. One-expression labels $\bullet \xrightarrow{x} \bullet$, expressing that x may have a value at the current program point, may be interesting for other applications but are irrelevant for the theory of aliasing, at least until it gets extended for object-oriented programming.

3.3 The semantics of an alias relation

If a is an alias relation on the set E of reference variables and expressions appearing in a program p , we may associate with a the assertion written a^- and defined as

$$a^- \triangleq \bigwedge_{[x, y] \notin a} x \neq y$$

In words: a^- is the assertion stating that no two references may be equal unless their pair appears in a . This notation reflects the conservative nature of the calculus: while the *presence* of a pair $[x, y]$ in a states that x *might* become aliased to y but does not imply that it will, its *absence* from a implies, for soundness, that x *will not* become aliased to y .

We also define the “quotient” a/x of an alias relation a in $E \leftrightarrow E$ by an element x of E (similar to the equivalence class of x in an equivalence relation) as the set containing all elements aliased to x plus x itself:

$$a/x \triangleq \{y: E \mid (y = x) \vee [x, y] \in a\} \quad /6/$$

3.4 Basic operation

Aliasing is not compositional, in the naïve sense of allowing the definition of a function *aliases* such that *aliases* (*p*) would determine the alias relation induced by the program *p* in terms of *aliases* (*p_i*) for components *p_i* of *p*. Consider

<i>p1</i> :	$x := y$
<i>p2</i> :	$z := x ; x := u$
<i>p1 ; p2</i> :	$x := y ; z := x ; x := u$

then *aliases* (*p1*) would be $\overline{x, y}$, *aliases* (*p2*) would be $\overline{x, u}$, and *aliases* (*p1 ; p2*) would be $\overline{y, z, x, u}$, which cannot be obtained by combination of the previous two since neither of them mentions *z*. Instead, the calculus works on formulae of the form

$a \models p$	/7/
---------------	-----

where *a* is an alias relation and *p* is a program component. /7/ denotes the alias relation that holds at the end of an execution of *p* started in a state where *a* held.

More precisely, both *a* and $a \models p$ are possibly conservative approximations of the actual alias relation. The semantics of the \models notation is captured by the following fundamental soundness requirement, expressed as a Hoare triple:

The alias calculus soundness condition	
For any relation <i>a</i> and any construct <i>p</i> :	
$\{a\} \ p \ \{(a \models p)\}$	/8/

This states that if we use *a* as a guarantee about pairs that will not be aliased on entry to *p*, the calculus yields a guarantee about pairs that will not be aliased on exit.

The rules of the calculus, as presented next, define $a \models p$ for every kind of instruction *p*. To be acceptable, any such definition must guarantee that if *a* is an alias relation so is $a \models p$. In addition, the soundness of the calculus requires a proof that every rule satisfies the fundamental soundness condition /8/. The present discussion does not include a complete proof but gives an example, for one of the rules, in section 4.12.

4 The basic calculus

The first level of the calculus relies on a simple programming language, E0. The following subsections introduce the constructs of E0, their informal semantics, and the corresponding alias calculus rules.

In E0, all variables denote references; the value of a reference is an object identifier. A formal definition of E0 appears below (4.12), but we will for the basic presentation rely on an intuitive understanding of the instructions.

4.1 Skip

It is convenient to include a null instruction **skip** with the rule

$$a \models \mathbf{skip} \quad = \quad a \quad /9/$$

(Shaded lines will signal rules of the alias calculus.)

4.2 Forget

If x is a variable, the notation **forget** x denotes an instruction that removes any association of x with any object. Corresponding programming language notations are:

$x := \mathbf{Void}$	-- Eiffel
$x = \mathit{null};$	-- C, Java etc.

(The reason for the special E0 syntax **forget** x is that experience has shown that using an assignment syntax, such that $x := \mathbf{Void}$, causes confusion with the regular form of assignment seen in 4.5 below.)

The rule is:

$$a \models (\mathbf{forget} \ x) \quad = \quad a \vdash \{x\} \quad /10/$$

The operator \vdash is defined as follows: $r \vdash A$, where r is a relation in $E \leftrightarrow E$ and A is a subset of E , is r deprived of any pair that involves a member of A as first or second element. Formally: $r \vdash A$ is $r - (A \times E)$. If r is an alias relation, so is $r \vdash A$. (The operator's definition will be extended in 6.5 to cover dot expressions.)

Imprecision: this rule introduces no imprecision.

Alias diagram: to carry out **forget** x on a diagram, remove x from all edge labels that included it; to maintain the canonical form, we must also remove any edge that as a result goes down to a one-element label, as in this example:



4.3 Creation

If x is a variable, the notation **create** x denotes an instruction that allocates a new object at a previously unused address. Corresponding programming language notation are:

create x	-- Eiffel
$x = \text{new Type_of_}x ();$	-- C, Java etc.

The effect on an alias diagram is the same as for **forget** x , and so is the rule:

$$a \models (\text{create } x) \quad = \quad a \vdash \{x\} \quad /11/$$

Imprecision: this rule introduces no imprecision.

The **forget** and **create** instructions have different semantics — one removes all associations of a given variable with any objects, the other associates it with a new object — but in the alias calculus they are governed by identical rules.

4.4 Cut

If x and y are variables, the notation **cut** x, y denotes an instruction that removes any aliasing between x and y . It does not correspond to any common instructions of programming languages but, as noted in 2.2, will serve as an essential escape mechanism to remove undesired cases of imprecision in the calculus. The constructs

check $x \neq y$ end	-- Eiffel
assert $x \neq y;$	-- JML, Spec#

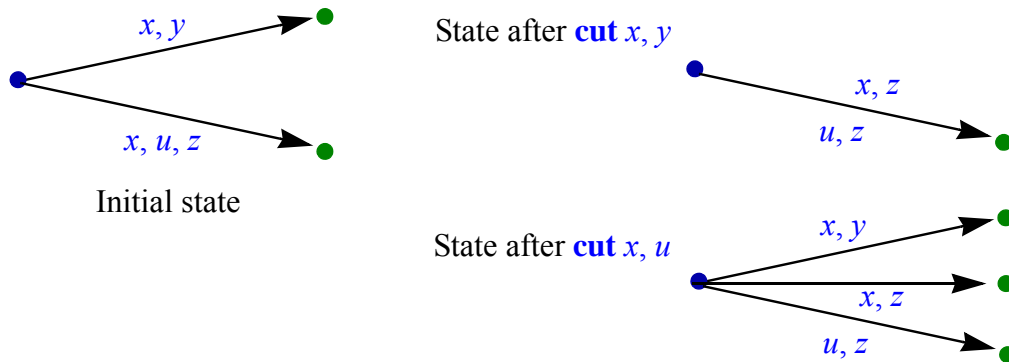
will be translated into **cut** x, y . (The semantics of **check** p **end** in Eiffel is that the program is only valid with a proof that p will always hold at the given program point; it is also possible for compilers that cannot perform such proofs to generate a run-time check that will stop the program if p does not hold. The rules for **assert** in Spec# and JML are similar.)

The alias calculus rule is:

$$a \models (\text{cut } x, y) \quad = \quad a - \overline{x, y} \quad /12/$$

Imprecision: this rule introduces no imprecision.

Alias diagram: to carry out **cut** x, y , remove any edge with label x, y ; replace any edge whose label includes x, y and a non-empty set A of other expressions by two edges, labeled x, A and y, A , to two separate nodes.



The need, in the second case, to replace an edge (and node) by two reflects the suggested practical use of **cut**: the operator lets us take advantage of finer-grain information, possibly coming from other sources, to improve the precision of the information provided by alias analysis. In the second case of the diagram above, the initial state conflated all of x , u , z into a single alias class; as we find out that x and u are not related after all, we separate these into two classes $\overline{x, z}$ and $\overline{u, z}$ listing z 's aliasing associations separately. The formal rule /12/ covers this semantics succinctly; it does not need to distinguish between the two cases illustrated by the diagram.

4.5 Assignment

The basic operation that creates alias pairs is assignment, written $x := y$. The rule is:

$$a \models (x := y) \quad = \quad \text{given } b \triangleq a \vdash \{x\} \text{ then} \quad /13/ \\ \frac{\quad}{\overline{b \cup (\{x\} \times (b/y))}} \\ \text{end}$$

The intuition behind this operator is that the assignment causes:

- Removal of any previous aliasing of x .
- Then, aliasing of x to y and to any other expression previously aliased to y .

Rule /13/ expresses this property. The relation b is a deprived of any pair involving x . The right side yields all the aliases not involving x , then adds the pairs $\overline{[x, u]}$ where u is in b/y , that is to say (/6/) either is y or was aliased to y in b , and applies the overline operator to symmetrize the relation.

Example 1: the value of $a \models (z := f)$, where a is

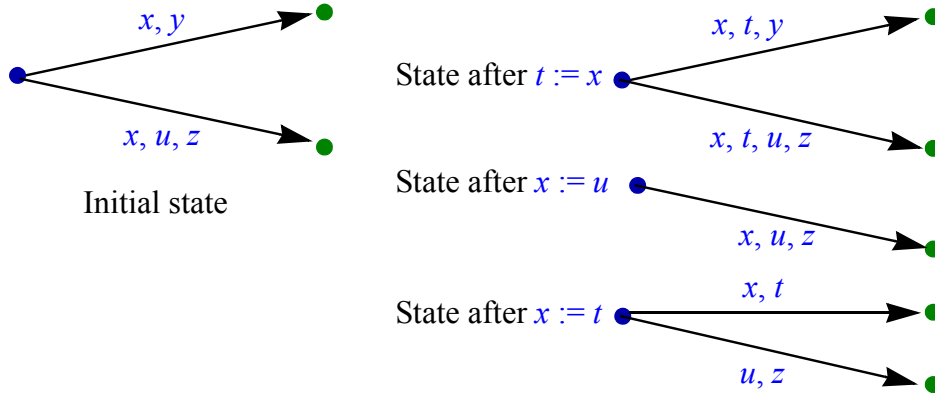
$$\overline{\overline{b, c, x, f, g, x, y, z}} \quad /14/$$

is (this example and all the following ones are as computed by the prototype implementation at se.ethz.ch/~meyer/down/alias.zip, on which the reader may try them):

$$\overline{\overline{b, c, x, f, g, x, z}}$$

where z has been removed from its previous association with y , then added to the associations of f .

Alias diagram: to carry out an assignment $x := y$ on an alias diagram, remove x from all edge labels (removing the edge if the label goes down to zero or one element); if y does not appear in any edge label, add a value node and an edge to it, labeled y ; then add x to any edge label containing y . Some examples:



4.6 Compound

If p and q are E0 instructions, the notation $p ; q$ denotes an instruction that executes p then q . The alias calculus rule is:

$$a \models (p ; q) \quad = (a \models p) \models q \quad /15/$$

If the other rules of the calculus guarantee that $a \models p$ is an alias relation whenever a is, this one also recursively yields an alias relation on the right side.

Imprecision: this rule introduces no imprecision.

Alias diagram: to carry out $p ; q$, apply the transformations associated with p , then apply to the resulting graph those associated with q .

4.7 Conditional

E0 has a conditional instruction of the form

then p else q end

where p and q are instructions. The informal semantics of this instruction is that it executes either p or q .

The rule is:

$$a \models \text{then } p \text{ else } q \text{ end} \quad = (a \models p) \cup (a \models q) \quad /16/$$

The \cup operator is here applied to two relations viewed as sets of pairs. As noted, $r1 \cup r2$ is an alias relation if both $r1$ and $r2$ are.

Imprecision: the conditional rule does not by itself introduce any imprecision, if we take the semantics of **then p else q end** to be that an execution can carry out either p or q . In the translation of an ordinary programming language to E0, the source instruction would be **if $cond$ then p else q end** for some condition $cond$. The condition is lost in translation; this may cause imprecision as in the earlier example (13/).

Example 2: the program

then $x := b$ else $x := f; z := y$ end /17/

yields, when applied to $\overline{b, c}, \overline{f, g}$, the alias relation $a = \overline{b, c, x}, \overline{f, g, x}, \overline{y, z}$ used as starting relation for the assignment example (14/).

Note on the example: the reader may wonder whether the assignment $z := y$ makes any sense without a prior assignment of a meaningful value to y . Such cases already arose in previous examples. For the alias calculus, however, this question need not alarm us, as it is a matter of convention for the underlying programming language. Some languages, such as the current void-safe version of Eiffel, guarantee that in any valid program y will automatically be initialized on first use to a legal address, denoting an object. Alternatively, we may take the convention that every example program in this article implicitly starts with a sequence of **create x** instruction, one for every variable x appearing in the program. Or we could pass on the requirement to the programmer by including a static rule that disallows access before creation, in which case (17/ is invalid.

Alias diagram: to carry out **then p else q end**, produce two diagrams by separately applying p and q to the original diagram. Then combine the diagrams by retaining all their value nodes and all their edges. The result correctly represents the effect of the conditional but may not be in canonical form; make it canonical following the procedure seen in 3.2.

4.8 Repetition

E0 has an instruction

p^n

where n is a natural integer. The semantics is that of **Skip** if $n = 0$ and otherwise, recursively, to execute $p^{n-1}; p$. Informally, this means n executions of p .

The instruction is not important by itself (as only a few programming languages support it, such as Fortran with its DO loop) but as a stepstone to the next construct, the loop instruction.

The rule is:

$a \models p^0$	$= a$		/18/
$a \models p^n$	$= (a \models p^{n-1}) \models p$	-- For $n > 0$	/19/

and is a direct consequence of the compound rule /15/.

Imprecision: the rule does not introduce any imprecision.

Examples 3 to 8: take $x := y; y := z; z := x$ for p and $\overline{c, y}, \overline{d, z}$ for a . Then:

$a \models p^0$	$= a$	$= \overline{c, y}, \overline{d, z}$
$a \models p^1$		$= \overline{c, x, z}, \overline{d, y}$
$a \models p^2$	$= a \models p^0$	$= \overline{c, y}, \overline{d, x, z}$
$a \models p^3$	$= a \models p^1$	$= \overline{c, x, z}, \overline{d, y}$
$a \models p^4$	$= a \models p^2$	$= \overline{c, y}, \overline{d, x, z}$
<i>etc.</i>		

The sequence oscillates indefinitely, for odd and even n , between the values of $a \models p^1$ and $a \models p^2$. This is as intuitively expected since p swaps the values of y and z .

4.9 Loop

The E0 instruction

loop p end

has the informal semantics of executing p repeatedly any number of times, including 0. Formally, if an instruction is defined as a relation between input and input states (see 4.12 below), then **loop p end** is simply $\bigcup_{n: \mathbb{N}} p^n$.

A first form of the loop rule follows from this definition:

$a \models \mathbf{loop\ } p \ \mathbf{end}$	$= \bigcup_{n: \mathbb{N}} (a \models p^n)$	/20/
--	---	-------------

Imprecision: the rule by itself does not introduce any imprecision. Imprecision may follow, however, from translating loop constructs as found in actual programming languages into the E0 form, since the translation will lose any information that the programmer or prover may have about the actual number of iterations, as might be deduced for example from the loop exit condition.

Theorem: the alias relation induced by a loop per /20/ is finite.

Proof: trivial since our alias relations are members of $\mathbb{P}(E \times E)$ for a finite set E (of variables and expressions appearing in a program), so they can only be finite.

This theorem, and the loop rule in its first form /20/, are not directly useful since they do not yield a practical way of computing $a \models \mathbf{loop\ } p \ \mathbf{end}$. A more interesting version of the theorem, the loop aliasing theorem, follows from the discussion of continuity appearing next, and yields the practical version of the loop rule, given as /25/ below.

4.10 Monotonicity and the loop aliasing theorem

To deal effectively with loops, and procedures as introduced next, we need structural properties. For any instruction p , we define monotonicity of the \models operator, with respect to the partial order relation \subseteq (here over relations, that is to say, subsets of $E \times E$), as the following property for any alias relations a and a' :

$$a \subseteq a' \Rightarrow (a \models p) \subseteq (a' \models p) \quad /21/$$

Alias monotonicity theorem: all rules given so far satisfy monotonicity.

Proof: the rules for the control structures — compound, conditional, repetition and loop — clearly preserve monotonicity if the constituent instructions satisfy it; so we must establish monotonicity for basic instructions. Since $a \models p$ is deduced from a , and $a' \models p$ similarly from a' , by some set of additions and removals of pairs, the proof must show that any pair added to a is also added to a' and that any pair removed from a' either was not in a or is also removed from a . The only direct source of additions is the assignment rule /13/; added pairs for the assignment $x := y$ include $[x, y]$, which will also be added to a' , and $[x, z]$ where a pair $[y, z]$ was in a , and hence in a' , so that the same pair will be added to a' . Removal of pairs occurs through the rules for **forget**, **create**, **cut** and assignment. In the first three cases the pairs marked for removal depend entirely on the instruction and not on a or a' : removing any of them from a' will remove it from a if it was there. In the assignment case, the pairs removed are of the form $[u, v]$ where either u or v is x ; if any such pair in a' is also in a , it will be removed from a . The rule also removes all reflexive pairs, but none of those comes from the original a or a' as they are alias relations.

The following properties are also of interest:

$$\begin{aligned} ((a \models p) \cup (a' \models p)) &= (a \cup a') \models p & /22/ \\ (a \cap a') \models p &= ((a \models p) \cap (a' \models p)) & /23/ \end{aligned}$$

In each case the left side is a subset of the right side as a consequence of the alias monotonicity theorem. The proof of the reverse inclusions follows, as for that theorem, from considering additions and removals for each kind of instruction.

The next theorem yields a practical way to compute the alias relation induced by a loop:

Loop aliasing theorem

For given p , let the sequence t be defined by $t_0 = a$ and $t_{n+1} = t_n \cup (t_n \models p)$.

There exists an integer N such that

- 1 For any $i < N$, $t_i \neq t_{i+1}$.
- 2 For any $i > N$, $t_i = t_N$.
- 3 $t_N = (a \models \mathbf{loop } p \mathbf{ end})$.

Proof: the first two properties are immediate:

- The sequence t_n is non-decreasing over a finite set, and hence has a fixpoint.

- A non-decreasing sequence might encounter two or more equal consecutive elements (a plateau) before it reaches its fixpoint. This, however, cannot happen for a sequence defined in the form $t_{n+1} = f(t_n)$ (here $t_{n+1} = t_n \cup (t_n \models p)$): if $t_N = t_{N+1}$, then $t_{N+2} = f(t_{N+1})$, also equal to $f(t_N)$ and hence to t_{N+1} and t_N ; all subsequent elements are equal as well. So the fixpoint is reached at the first N such that $t_N = t_{N+1}$; this is the N of the theorem.

To prove property 3, we will prove that t_n is the same sequence as the sequence s_n defined as

$$s_n \triangleq \bigcup_{i: 0..n} (a \models p^i) \quad /24/$$

This will give us the desired result since $a \models \mathbf{loop\ } p \ \mathbf{end}$, defined in /20/ as $\bigcup_{n:N} a \models p^n$, is also as a consequence $\bigcup_{n:N} s_n$; since $s_n \subseteq s_{n+1}$ for all n , the fixpoint of the sequence (the first N such that $s_N = s_{N+1}$) will, if the sequences s_n and t_n are the same, yield $a \models \mathbf{loop\ } p \ \mathbf{end}$.

The proof that the sequences are the same uses induction. First, $s_0 = t_0 = a$ and $s_1 = t_1 = (a \cup (a \models p))$. (The second part of the proof needs both base steps.) For the induction step, we prove separately that $s_{n+1} \subseteq t_{n+1}$ and $t_{n+1} \subseteq s_{n+1}$. For the first property we expand the definition:

$$s_{n+1} = s_n \cup a \models p^{n+1}$$

Since $s_n = t_n$ by the induction hypothesis and $t_n \subseteq t_{n+1}$ by the definition of t , it suffices to prove that $a \models p^{n+1} \subseteq t_{n+1}$. By the definition of repetition, $a \models p^{n+1} = (a \models p^n) \models p$. We note that $a \models p^n \subseteq s_n$ by the definition of s_n /24/, so $a \models p^n \subseteq t_n$ by the induction hypothesis. This implies by monotonicity that $((a \models p^n) \models p) \subseteq (t_n \models p)$ and hence (by the definition of the sequence t_n) that $((a \models p^n) \models p) \subseteq t_{n+1}$. This completes the proof that $s_{n+1} \subseteq t_{n+1}$.

For the induction step in the reverse direction, we expand the other definition :

$$\begin{aligned} t_{n+1} &= t_n \cup (t_n \models p) && \text{-- By the definition of } t_n \\ &= s_n \cup (s_n \models p) && \text{-- By the induction hypothesis} \end{aligned}$$

Since $s_n \subseteq s_{n+1}$ it suffices to prove that $(s_n \models p) \subseteq s_{n+1}$. Since we have two base steps ($n = 0$ and $n = 1$), we may assume $n > 1$ and expand s_n as $s_{n-1} \cup (a \models p^n)$, so that by /22/ $s_n \models p$ is $(s_{n-1} \models p) \cup (a \models p^{n+1})$; since the first operand is $t_{n-1} \models p$ by the induction hypothesis and hence a subset of t_n (which is also s_n), both terms are subsets of s_{n+1} .

As a consequence of this theorem we will use the following version of the loop rule:

$$\begin{aligned} a \models \mathbf{loop\ } p \ \mathbf{end} &= t_N && /25/ \\ &&& \text{-- For the first } N \text{ such that } t_N = t_{N+1}, \\ &&& \text{-- with } t_0 = a \text{ and } t_{n+1} = t_n \cup (t_n \models p). \end{aligned}$$

Example 9: a loop with the same body as in the repetition example

```
loop  $x := y ; y := z ; z := x$  end
```

and started with the same initial alias relation $a = \overline{c, y}, \overline{d, z}$ reaches its fixpoint at t_2 :

```
 $t_0 = a = \overline{c, y}, \overline{d, z}$ 
 $t_1 = \overline{c, x, z}, \overline{c, y}, \overline{d, y}, \overline{d, z}$ 
 $t_2 = \overline{c, x, z}, \overline{c, y}, \overline{d, x, z}, \overline{d, y}$ 
 $t_3 = t_2$  -- etc. (all subsequent values equal to  $t_2$ ).
```

In this example, the sequence $a | = p^n$ did not converge, as we saw in 4.8. But the loop aliasing theorem tells us that the sequence t_n always reaches a fixpoint finitely.

4.11 A more intricate example

Example 10: as a more extensive application of the E0 calculus, involving instructions of all the kinds encountered so far, consider the following program p (semicolons omitted at end of lines):

```
then  $x := y$  else  $x := a$  end
then cut  $x, y ; z := x$  else end
 $g := h ; x := y ; z := a ; b := x$ 
loop  $e := f ; a := e$  end
loop
  then  $c := b ; a := f ; g := x$  else  $c := a ; a := g$  end
   $f := x$ 
end
 $b := z ;$  forget  $b ; a := e ;$  create  $z ; a := h ;$  cut  $a, g ;$  create  $x$ 
```

The value of $\emptyset | = p$ is: $\overline{a, c, h}, \overline{c, e, f}, \overline{c, f, g, y}, \overline{c, g, h}$.

4.12 Formalizing E0 and soundness

This subsection does not introduce any new properties of the alias calculus but shows how the calculus can be proved in reference to a formal definition of the E0 language. Readers interested mostly in the rules of the alias calculus can skip to section 5.

An E0 program may be defined as a relation in $State \leftrightarrow State$. A deterministic language would use functions, possibly partial, rather than relations; non-determinism keeps the language definition simple, in particular for the loop construct.

A state s is characterized by:

- A set of variables that have a value in that state: $s.def$ (a member of $\mathbb{P}(Variable)$).
- A set of addresses allocated in that state: $s.addr$ (a member of $\mathbb{P}(Address)$, assuming a suitable set $Address$).

- The values of the variables in the state, as represented by a function $s.value$, a member of $Variable \rightarrow Address$ (using \rightarrow for the set of possibly partial functions), where $\mathbf{domain}(s.value) = \{v: Variable \mid v \in s.def\}$.

To define a state s , it suffices to give $s.def$, $s.addr$ and $s.value$.

To define E0 formally we specify each instruction as a relation in $State \leftrightarrow State$, by considering in each case an arbitrary state σ and stating the properties of states σ' that may result from applying p . For example, in the case of **skip** (the identity relation on $State$), $\sigma' = \sigma$.

For the instruction **forget** x , the definition is: $\sigma'.def = \sigma.def - \{x\}$; $\sigma'.addr = \sigma.addr$; $\sigma'.value(y) = \sigma.value(y)$ for $y \neq x$.

For **create** x , for some na in $Addresses$ such that $na \notin s.addr$: $\sigma'.def = \sigma.def \cup \{x\}$; $\sigma'.addr = \sigma.addr \cup \{na\}$; $\sigma'.value(y) = na$; $\sigma'.value(y) = \sigma.value(y)$ for $y \neq x$.

For $x := y$: if $y \notin s.addr$, as for **forget** x ; otherwise: $\sigma'.def = \sigma.def \cup \{x\}$; $\sigma'.addr = \sigma.addr$; $\sigma'.value(x) = \sigma.value(y)$; $\sigma'.value(z) = \sigma.value(z)$ for $z \neq x$.

For the compound $p ; q$: what this notation means as a mathematical convention, taken to denote composition of relations in the order given (the same as $q \circ p$).

All the elementary constructs defined so far are functions (deterministic). Non-function relations (representing possible non-determinism) may arise with:

- Conditional: **then** p **else** q **end** is defined simply as another notation for $p \cup q$.
- Loop: **loop** p **end** is defined as $\bigcup_{n:\mathbb{N}} p^n$. The term p^n (corresponding to the E0 repetition construct) retains its definition from mathematics: $p^0 = \mathbf{skip}$, $p^{n+1} = (p^n ; p)$.

In this framework, every state induces an alias relation defined as

$$\mathbf{aliases}(\sigma) \triangleq \{[x, y] \mid x \in \sigma.def \wedge y \in \sigma.def \wedge \sigma.value(x) = \sigma.value(y)\}$$

An earlier formula /8/ defined soundness in an axiomatic semantics style. For a language such as E0, where instructions and programs are defined directly as relations, we may use the following definition of the soundness property, for any instruction p :

$$[\sigma, \sigma'] \in p \Rightarrow \mathbf{aliases}(\sigma') \subseteq (\mathbf{aliases}(\sigma) \mid p) \quad /26/$$

As an example of soundness proof, consider **forget** x . For a given σ , the above definition of the **forget** instruction tells us that there is only one σ' and that a pair $[y, z]$ is in $\mathbf{aliases}(\sigma')$ if and only if $y \neq x$, $z \neq x$ and $\sigma.value(y) = \sigma.value(z)$. The pair is also in $\mathbf{aliases}(\sigma) \mid \mathbf{forget} x$ since the **forget** rule /10/ defines $\mathbf{aliases}(\sigma) \mid \mathbf{forget} x$ as $\mathbf{aliases}(\sigma) \setminus \{x\}$.

In this example the \subseteq relationship of the soundness requirement /26/ is actually an equality. This is also the case with other constructs seen so far since, as noted, they do not introduce imprecision.

Soundness proofs should similarly be provided for every instruction, although they do not appear in the present article.

5 Introducing procedures

Our next language, E1, simply adds to E0 the notion of procedure, without arguments. A procedure p is defined by a program name, written $p.name$, and an instruction, written $p.body$. E1 has a new instruction, **call** p , where p is a procedure; the effect is to execute $p.body$. (In a directly usable programming language the concrete syntax would use **call** pn where pn is $p.name$.) A program is defined by a non-empty set of procedures and the name of one of them, designating it as the main procedure.

The rule for the call instruction is:

$$a \models \mathbf{call} \ r \quad = \quad (a \models r.body) \quad /27/$$

and for a program pr with main procedure $Main$:

$$a \models pr \quad = \quad (a \models Main.body) \quad /28/$$

which will be used in practice with \emptyset for a , assuming every program starts with an empty alias relation.

In the absence of mutually recursive procedures, computing the alias relation of a program can simply proceed as in the previous examples: for every program element p , starting with the entire program, apply the corresponding alias calculus rule, which expresses $a \models p$ in terms of $a' \models p'$ for sub-elements p' of p ; the process terminates when applied to atomic elements such as assignments. This scheme no longer directly works for a program that includes mutually recursive procedures, since the computation of $a \models r.body$ through the call rule /27/ may lead to a new evaluation of $a \models \mathbf{call} \ r$. To obtain a directly applicable process, we note that if a program consists of a number of procedures r_1, r_2, \dots, r_n , and use the notation $b_i(a)$ for $a \models r_i.body$, we may write the application of the call rule to any one of them, expanding $a \models r_i.body$, as

$$b_i(a) \quad = \quad AL_i(b_1(f_{1,i}(a)), b_2(f_{2,i}(a)), \dots, b_n(f_{n,i}(a)))$$

where all the functions involved, AL_i and $f_{j,i}$, deduced from applying the rules of the calculus to the text of b_i , are monotone. If r_i is the main procedure, defining the alias relation induced by the whole program, computing $b_1(\emptyset)$ will give us, in the resulting b vector, the alias relation at the exit point of every procedure (which is where we need it to apply axiomatic semantics, for example in weakest-precondition style). Since all functions involved are monotone and the set of relations is finite, standard reasoning shows that starting with empty relations for all the b_i and iterating will reach a fixpoint finitely, yielding the desired result. The prototype implementation directly applies these ideas, as illustrated by the following example.

Imprecision: by itself this rule introduces no imprecision. Translations from programming languages will, however, cause imprecision because the procedure mechanism does directly not support arguments, local variables and return values. For a typical procedure

```
p (a: SOME_TYPE) do ... end
```

the translation will replace *a* by a variable, and understand a call *p* (*x*) as the E1 instructions

```
a := x  
call p
```

The same scheme applies to local variables, and (since the language only supports procedures) to the result value of a function. As a consequence, the translation will lump together, for the computation of aliases, the values of local variables, results and formal arguments that belong to different recursive incarnations of a given recursive routine (or to concurrent executions of that routine in different threads).

Example 11 (in this example and the following ones the starting alias relation is empty): we consider the recursive procedure

```
procedure Main  
  then  
    x := y  
  else  
    x := a ; call Main  
  end  
end
```

The resulting alias relation is just $\overline{x, y}$: the second branch of the conditional can never contribute anything.

Example 12: If we reverse the order of the instructions in the else clause of the previous example (giving **call** *p* ; *x* := *a*), we get $\overline{a, x}, \overline{x, y}$.

Example 13: the following are mutually recursive procedures (still simple, to enable intuitive manual verification of the result):

```
procedure Main  
  then x := y else x := a ; call q end  
end  
procedure q  
  x := b ; then call Main else a := c end  
end
```

The result, with *Main* as the main procedure, is $\overline{a, c}, \overline{b, x}, \overline{x, y}$. In particular, *x* can get aliased to *a* and *a* to *c*, but not *x* to *c*.

Example 14: another case of mutually recursive procedures:

```

procedure Main
  then  $x := y$  else  $x := a$  end
  then cut  $x, y ; z := x$  else end
  then call  $q$  else  $g := h$  end
   $x := y ; z := a ; b := x$ 
  loop
     $e := f$ 
    then  $a := e$  else end
  end
  then  $c := b ; a := f ; g := x$  else
    loop  $c := a ; a := g$  end
    call Main
  end
   $f := x ; b := z ;$  forget  $b ; a := e ;$  create  $z ; a := h$ 
  cut  $a, g ;$  create  $x$ 
end
procedure  $q$ 
  then  $m := n$  else  $m := h ;$  call Main end
end

```

The result is $\overline{a, h, m, c, e, f, g, y, m, n}$. This example is not representative of any actual program but illustrates the application of the calculus to procedures with a complex recursion and control structure.

6 The object-oriented calculus

The next and last language level, E2, introduces object-oriented mechanisms. E2 is sufficient powerful to support applying the calculus to a modern object-oriented language such as Eiffel, Java or C#. The relevant part of object technology here is the dynamic object model: dynamic object creation, pointers or references (we will consider the two terms synonymous), and the possibility for objects to contain pointers to other objects. This last facility is the only novelty of E2's dynamic model, since E0 and E1 already offered the first two.

Other object-oriented mechanisms such as inheritance and genericity have only marginal influence on aliasing.

6.1 New language concepts

Making E2 support object-oriented programming means adding three language concepts:

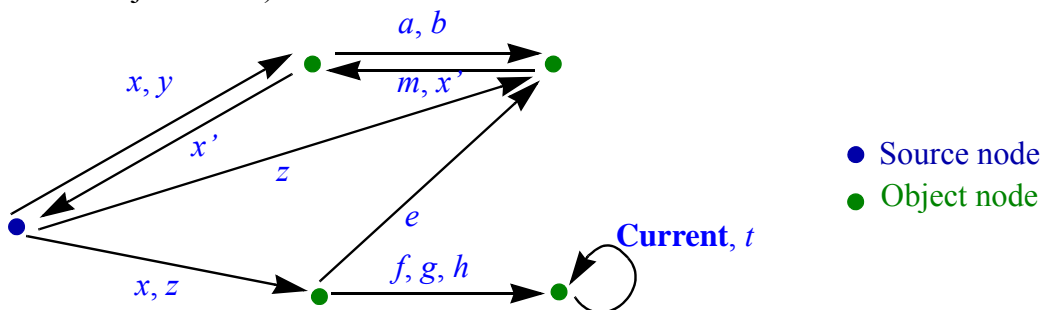
- Qualified expressions, such as $x.y.z$, which can be used as sources of assignments, as in $u := x.y.z$.
- Qualified calls, such as $x.f(v)$; as before we will limit ourselves to argument-less procedures and take care of argument passing through assignments.

- The notion of current object (**Current** in Eiffel, **this** in C++ and Java, **self** in Smalltalk). This is the central concept of object technology, giving rise to the “*general relativity*” principle of O-O programming: every operation is relative to a current object; starting a qualified call $x.f(v)$ makes a new object (the object attached to x) current; ending such a call restores the previous current object as current.

We will not directly consider qualified assignments of the form $x.a := v$ permitted by programming languages such as Java, C# and C++. It may be possible to include qualified assignments directly into the theory, a task that the present article does not undertake (as a matter of principle, since qualified assignments fly in the face of all principles of software engineering, and even the designers of languages that include this mechanism advise against using it); it happily leaves it for other authors to solve. The omission of this mechanism in the theory and calculus as described here has no practical consequence on the application to the relevant programming languages, since it suffices to assume a pre-processing step that translates all qualified assignments $x.a := v$ into qualified calls to setting procedures, such as $x.set_a(v)$.

6.2 Object-oriented alias diagrams

E2 alias diagrams still have a source node, which now represents the current object, but that node no longer has any special property; edges can exist between value nodes (from now on called object nodes):



As the example suggests, cycles are now possible (between objects nodes only). As we will see, cycles arise as a result of passing arguments to qualified calls. The new forms of expressions appearing in the figure, **Current** and “inverted variables” such as x' , will be explained shortly.

An object node represents a set of possible objects, all of the same type (class); the interpretation of an edge with labels x, y, \dots between two object nodes, representing sets of objects $OS1$ and $OS2$, is that every object in $OS1$ may have reference fields to an object in $OS2$; since in typed object-oriented programming every field of an object corresponds to an attribute (also called “member variable” or “data member” in various O-O languages) in the relevant class, the fields involved are those corresponding to attribute names x, y, \dots

One-expression edge labels $\bullet \xrightarrow{x} \bullet$, previously discarded, are useful for O-O alias diagrams. Also, we no longer systematically remove the end node when we remove an edge, but only do so if no other edge leads to that node. (This property reflects the need for garbage collection in an object-oriented model.)

The variables appearing in labels represent attributes from the corresponding class. In the figure, x , y and z are attributes of the class of the current object; e , f , g and h are attributes corresponding to the class covering the object in the middle-bottom node. The calculus does not need information about the classes; we assume that it is applied to a type O-O language after type checking, so that every attribute name refers unambiguously to a class. This convention is particularly important in Eiffel where style rules suggest the systematic use, for consistency, of a small set of feature names such as *first and* item. In the application of the calculus to a specific programming language, a good convention might be to identify the class as part of the attribute name, as in *item_{LIST} item_{CELL}* etc. We will need no such convention here; note in particular that the leftmost and middle-bottom nodes in the last figure might correspond to objects of the same type or different types.

The other major innovation of the E2 calculus is the kind of possibly aliased expressions (the set E of earlier discussions) under consideration. In addition to single variables as before, expressions now include three more variants:

- The special expression **Current** represents the current object (relative to any node). Informally, **Current** denotes a link from a node to itself, as in the bottom-right node of the last figure.
- For any variable x , the **inverse** of x is written x' . Informally, consider a call $x.r$, executed on behalf of a certain client object, which applies r to a supplier object referenced by x ; then x' represents a reference back from the supplier to the client. It will appear in edges between the corresponding nodes, as in the preceding figures. Together with **Current**, the inversion operator is the reason why E2 graphs, unlike E0 and E1 graphs, may be cyclic.
- Finally, E2 supports dot expressions of the form $x.y.z\dots$

The presence of dot expressions gives an alias diagram a richer meaning: aliases arise not only from edges but also from *paths* in the diagram. The rule is that if two paths have the same starting node and the same ending node, the corresponding dot expressions are aliased. Consider for example, in the last diagram, the edge labeled z from the source node to the top-right node; it implies that z is aliased to $x.a$, $x.b$, $y.a$, $y.b$ (paths through top nodes) as well as $x.e$ and $z.e$ (bottom paths).

6.3 Formal model

Adapting the previous formal model (4.12) for E3 involves changing the representation of states and the signature of instructions. The state now involves a set of objects, where each object may contain references to other objects. An instruction, previously a relation in $State \leftrightarrow State$, now has the signature $Object \rightarrow State \leftrightarrow State$; the use of an *Object* as the first argument reflects the notion of current object and the principle of general relativity.

The full refinement of the formal model, and the corresponding proofs of soundness for the remaining rules given below, belong in another article.

6.4 Dot expression properties

For simplicity it is convenient to add the dot to the calculus as an operator on variables and expressions representing paths: if v is a variable and e an expression $x.y.z\dots$, we write $v.e$ to denote the path $v.x.y.z\dots$ and extend this notation to two expressions, writing $e.f$ for the concatenation of e and f .

The following fundamental property, reflecting the preceding observation on alias diagrams, characterizes the semantics of aliasing with dot expressions:

Dot completeness

An alias relation a involving dot expressions must satisfy, for any expression $e1, e2, f1$ and $f2$:

$$[e1, e2] \in a \wedge [f1, f2] \in a \Rightarrow [e1.f1, e2.f2] \in a \quad /29/$$

This requirement is added to the basic definition of alias relations as symmetric and irreflexive (3.1). If a is a symmetric and irreflexive relation, a^* will denote the smallest symmetric and irreflexive relation that includes a and satisfies dot completeness. For example if a is $\underline{x, y}, \underline{x.u, v.y}$, then a^* adds the pairs $[y.u, v.y], [x.u, v.x]$ (symmetrized).

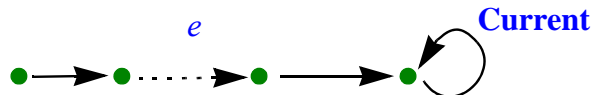
In the dot calculus, **Current** plays the role of zero element and variable inversion the role of the inverse operation. For any expression e (including a single variable) and any variable x :

Current.e	=	e	/30/
$e.$ Current	=	e	/31/
$x.x'$	=	Current	/32/
$x'.x$	=	Current	/33/

and as a consequence, for non-empty e :

$x.x'.e$	=	e	/34/
$x'.x.e$	=	e	/35/

/30/ and /31/ express that **Current** always represent a link to the current node. Note that the interpretation of **Current**, like everything else in the general relativity of object-oriented programming, pertains to an object and the corresponding class; /31/ describes a situation such as

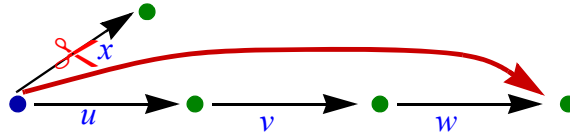


where the various nodes involved might correspond to different classes. **Current** is really **Current_C** for some class C . Clearly, $e.$ **Current_C** makes sense only if C is the class of the objects reached by e (the rightmost node in the figure); the alias calculus need not concern itself with this question, since we assume it is applied to type-checked programs.

In this framework, the alias calculus needs only two more rules to account for object-oriented programming: an adaptation of the assignment rule to account for multidot sources; and a rule for qualified calls **call $x.r$** .

6.5 Dot expressions as sources of assignments

In an assignment $x := y$, the source expression y may now be a multidot expression, such as $u.v.w$. An illustration with an example alias diagram (in this case with no aliasing) is:



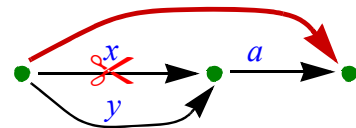
Only a small adaptation is needed to the original assignment rule /13/. In fact the rule itself does not change:

$$a \models (x := y) = \text{given } b \triangleq a \vdash \{x\} \text{ then } \frac{b \cup (\{x\} \times (b/y))}{\text{end}} \quad \text{-- Same as /13/}$$

but the operator \vdash must account for dot operators. The original definition (4.2) was that $r \vdash A$ is r deprived of any pair that involves a member of A . The revised definition (which covers the previous one for non-dot expressions) also removes from r any multidot expression whose first component (in the sense of u in $u.v.w$) is in A .

As a consequence, the set $\overline{b/y}$ (as used in the last set of pairs, $\{x\} \times (b/y)$, added to the relation on the second line above) may be empty, in which case $\{x\} \times (b/y)$ is itself empty. This reflects an important practical property: while in the non-O-O calculus an assignment $x := y$ always adds the pair $[x, y]$ to the alias relation, this is not necessarily the case with dot expressions. In the assignment

$$x := x.a$$



we should **not** alias x to $x.a$! This assignment removes all aliases of x , and creates no new aliasing unless x was previously aliased to some other expressions; then for every such expression y , it aliases x to $y.a$.

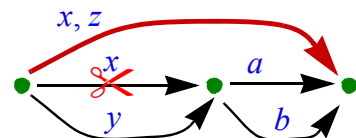
These observations do not rule out the possibility for x to become aliased to $x.a$; although such a situation cannot be the result of the assignment above, it will happen if a is aliased to **Current**.

The rule captures all these cases.

Imprecision: the rule introduces no imprecision.

Example 15: the following program uses dot expressions as assignment sources:

$$\begin{aligned} x := y ; a := b \\ z := x.a ; x := x.a \end{aligned}$$



The result (if we only include pairs that involve at least one non-dot expression) is $\underline{a, b, x, y.a, z, x, y.b, z}$.

6.6 Qualified call

The last remaining construct is the qualified procedure call **call** $x.r$. To handle it in the alias calculus, we need the following notation: if a is a relation (in our examples, an alias relation), $x \blacksquare a$ denotes the relation containing all pairs $[x.e, x.f]$ such that a contains $[e, f]$.

In a naïve approach to handling $x.r$, we would note that if a call to r (unqualified) aliases e to f then a call to $x.r$ aliases $x.e$ to $x.f$. Then $a \models x.r$ would be $x \blacksquare (a \models r)$. This does not, however, capture the possible changes to aliasing on the side of the client (the object on whose behalf the call $x.r$ is made). Consider for example, in an object-oriented programming language, the instructions

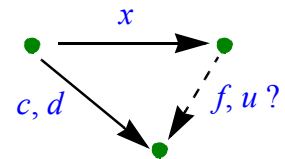
```

c := d
x.r(c)
```

with

```

r(u: T)
  do
    f := u  -- f is an attribute of the enclosing class.
  end
```

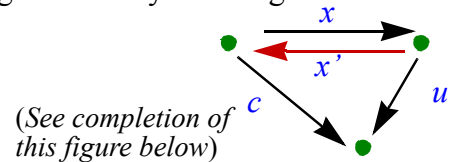


As usual, the alias calculus sees a call $x.r$ without arguments, whose execution starts with an assignment $u := f$ of the actual argument to the attribute representing the formal argument. The tentative rule would give us the (symmetrized) pairs $[c, d]$, $[f, u]$ and $[x.f, x.u]$, which are correct, as well as $[u, c]$ and as a result $[f, d]$ which are meaningless since they involve attributes applicable to different objects (and possibly classes). It misses, on the other hand, the aliasing of $x.f$ to c and d . It is unsound.

Obtaining a sound rule requires the inversion operator. The translation into E2 from an actual object-oriented programming language where procedures may have arguments will use the following convention (not part of the alias calculus, but necessary for an understanding of the rule): translate $x.r(c)$, where the corresponding formal argument in r is u as above, into **call** $x.r$, and add at the beginning of r 's body the assignment

```

u := x'.c
```



This convention explains the role of inverted variables such as x' : in a qualified call, provide a link back to the client, to enable the supplier, if needed, to update references on the client side — a principal property, although one fraught with obvious risks (aliasing risks in particular), of the object-oriented style of programming.

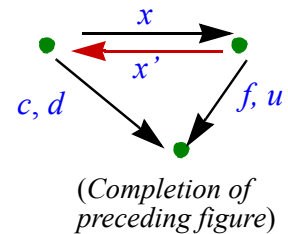
These observations also explain the inversion rules /32/ and /33/: $x.x'$ is **Current** (for the client) and $x'.x$ is **Current** (for the supplier).

The following, sound version of the rule describes the correct semantics of aliasing for qualified calls:

$$a \models \mathbf{call} \ x.r \quad = \quad x \cdot ((x' \cdot a) \models \mathbf{call} \ r) \quad /36/$$

(The last part, $\mathbf{call} \ r$, can also be written $r.body$ from the unqualified call rule /27/.)

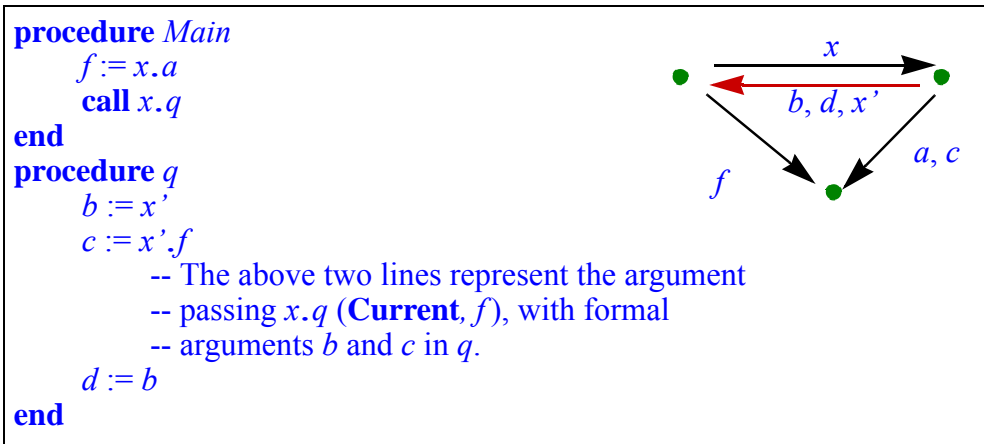
The rule works as follows. To compute the aliasings induced by $\mathbf{call} \ x.r$ in the aliasing environment a , we need to compute the aliasing induced by a simple unqualified call $\mathbf{call} \ r$; not in a , however, as a is relative to the client object, but in the view that the supplier object (corresponding to the target x) has of a . This view is $x' \cdot a$, with both elements of every pair in a prefixed by the inverted variable x' , a back pointer giving access to the client. This means in particular that if r executes $f := u$ where u is aliased to an actual argument c , known in the routine as $x'.c$, then f will get aliased to $x'.c$. The resulting alias relation, meaningful in the environment of the supplier, is $a' = ((x' \cdot a) \models \mathbf{call} \ r)$. In the end, however, we need to interpret the result back in the environment of the client, which knows the supplier as x ; so we use $x \cdot a'$, prefixing both elements of every pair in a' by x . If such an element is of the form $x'.e$, this prefixing will yield just e , since the dual rule /32/ tells us that $x'.e = \mathbf{Current}$. In the example, the pair $[x.f, x'.c]$ in a' will give $[x.f, c]$, and as a consequence $[x.f, d]$, in a ; this is the proper result as illustrated.



Thus we are permitted to prove that the unqualified call creates certain aliasings, on the assumption that it starts in its own alias environment but has access to the caller's environment through the inverted variable, and then to assert categorically that the qualified call has the same aliasings transposed back to the original environment. This change of environment to prove the unqualified property, followed by a change back to the original environment to prove the qualified property, explains well the aura of magic which attends a programmer's first introduction to object-oriented programming.

In the case of recursive or mutually recursive procedures, the qualified rule /36/ invalidates the finiteness arguments that the E1 discussion used to show the existence of a fixpoint reached finitely: every alias pair $[m, n]$ created by $\mathbf{call} \ r$ will yield (in the absence of x' aliasing) a pair $[x.m, x.n]$, increasing the dot count of both elements by one; with recursion the count would grow unbounded. This possibility causes no practical problem, however, since the basic assumption of the theory of aliasing (2.3) is that it only considers expressions that actually appear in a program. So it suffices to limit application of rule /36/ to alias relations a whose dot count is no greater than the maximum dot count of expressions in the program, defining the dot count of a pair of expressions as the maximum of the dot counts of its elements, and the dot count of a relation as the minimum dot count of its pairs. (The precise argument is more subtle, since in principle two expressions of the program could become aliased as a result of rule /36/ aliasing each of them to an expression not appearing in the program and having a dot count higher than any that will be computed using the limited rule. It is easy to see, however, that this case is impossible.)

Example 16: the following program includes a qualified call $x.q$, actually representing a call with arguments, $x.q$ (**Current**, f).



The resulting alias relation is $\overline{\text{Current}, x.b, x.d}, \overline{f, x.a, x.c}, \overline{x.b.f, x.c}$. As appropriate, it only includes aliases reachable from the node representing the current object (the top-left node in the figure). An aliasing such as a, c , which applies to another node (the rightmost one in the figure, which represents the target of the call $x.q$) appears in its form relative to the current object node, as $x.a, x.c$.

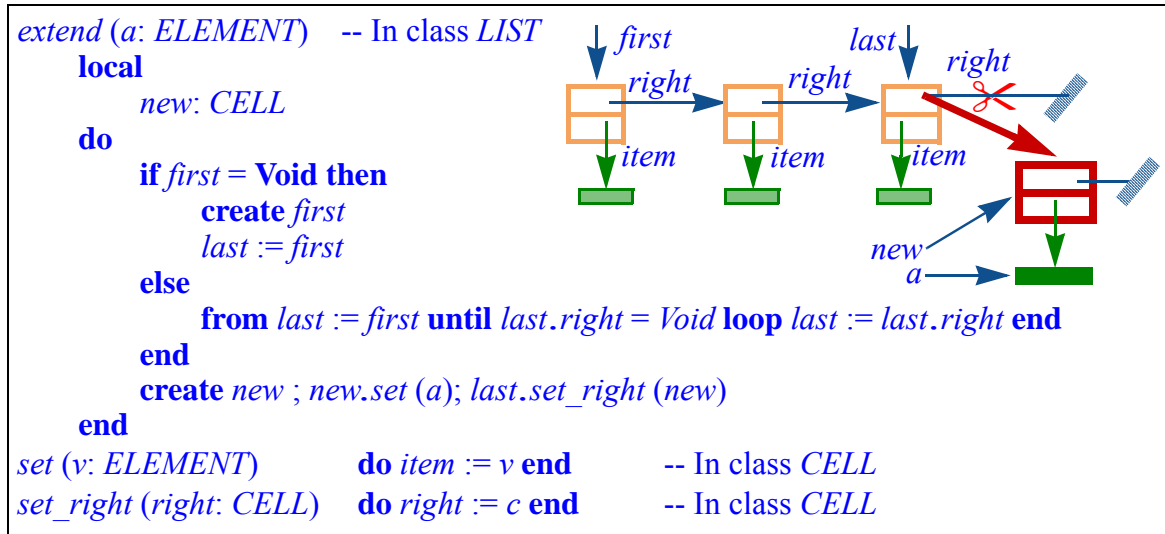
In the technique used so far, the assignments representing the argument passing appear in the called procedure (see the first two lines of q), rather than in the caller. If a procedure is called from several places, the corresponding assignments should then appear as separate branches in **then ... else ...** conditionals. This method is not good for modularity. It is preferable to introduce in the procedure a variable r_client (where r is the procedure name) representing the client, and set this variable *on the client side* prior to the call. With this approach, we may rewrite the above example as



Example 17: the program shown above gives the same alias relation as example 16, extended with properties of q_client .

6.7 Aliasing among list structures

Example 18: for the final example (this variant and the next), consider the list manipulation program mentioned in the introduction. To model the *LIST* and *CELL* procedures



we use the following E2 procedures:

```

procedure extend      -- In LIST
  a := extend_client'.el
then
  create first ; last := first
else
  last := first
end
loop last := last.right end
create new ; call new.set ; call last.set_right
end
procedure set          -- Called from only one place, with target new and argument a.
  item := new'.a
end
procedure set_right   -- Called from only one place, with target last and argument new.
  right := last'.new
end

```

Assume two separate lists *x* and *y*, to which we may add elements to our heart's content:

```

procedure build
  -- The two lines below could also be in separate branches of a then ... else.
  extend_client := x ; loop create el ; call x.extend end
  extend_client := y ; loop create el ; call y.extend end
end

```

Then we repeatedly access arbitrary elements of either list:

```

procedure Main
  call build
   $f := x.first ; g := y.first$ 
  loop
    then  $f := f.right$  else  $g := g.right$  end
  end
end

```

The alias relation (as obtained from running this example in the implementation, and removing the `extend_client` variable from the output) is:

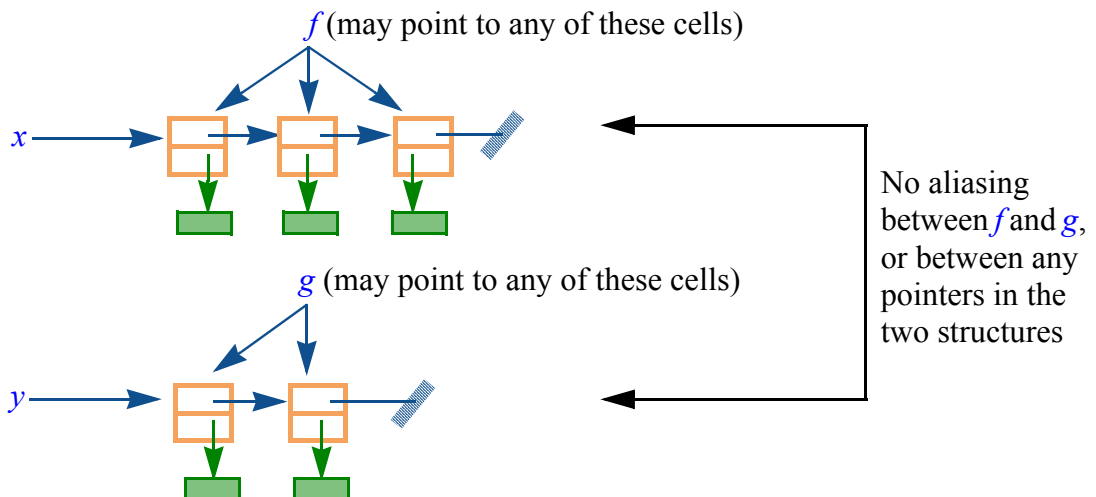
```

 $\overline{f, x.first, x.last}, \overline{f, x.first.right, x.last}, \overline{f, x.first.right.right, x.last}, \overline{f, x.last.right},$ 
 $\overline{f, x.last.right.right}, \overline{g, y.first, y.last}, \overline{g, y.first.right, y.last},$ 
 $\overline{g, y.first.right.right, y.last}, \overline{g, y.last.right}, \overline{g, y.last.right.right}, \overline{x.a, x.new.item},$ 
 $\overline{x.last.right, x.new}, \overline{x.a, x.new.item}, \overline{y.a, y.new.item}, \overline{y.last.right, y.new}$ 

```

The full relation, as noted, would be infinite; it includes for example all pairs of the form $[x.first.right\dots, x.last]$ with an arbitrary number of “.right” after $x.first$. As discussed in 6.6, the application of the theory to a particular annotated program breaks off at the highest dot length of expressions found in the program. To run the examples, the current implementation sets this maximum to three dots, as illustrated in the above result.

The most important property of that result is that the relation does **not** include the pair $[f, g]$, showing that no pointer in either list can ever become attached to a cell of the other:



Example 19: Add the assignment $x := y$ at the beginning of `Main`; keep the rest of example 18 unchanged. The resulting alias relation now includes $f, g, x.first, y.first, f, g, x.first.right$ etc. (run the implementation to see the full list). The important property is that now, as a result of this single change, f can be aliased to g .

7 Prototype implementation

The prototype implementation is stand-alone, rather than integrated into the compiler of a programming language. It is written in Eiffel; mechanisms of inheritance (particularly multiple inheritance), genericity and contracts have proved essential to the prompt completion of this implementation. Using an imperative language (including numerous mechanisms found in functional languages) was a key factor in this process; in particular, many delicate decisions involved when to duplicate a data structure, such as the representation of an alias relation, and when simply to update it.

The implementation makes it possible to write an E2 program and produce its alias relation in canonical form, as illustrated by the examples of this article. All the examples are part of the implementation and can be tried in the downloadable version.

The response for these examples is immediate, but no complexity analysis has been performed to explore scalability to large programs.

8 Application to a programming language

The translation from an actual programming language involves the steps discussed earlier: ignoring conditions of conditionals and loops; replacing functions by procedures; replacing arguments, local variables and function results by attributes; associating inverted variables with actual arguments of qualified calls.

9 Open problems

A number of problems remain to be addressed:

- Although the existing implementation provides a convincing proof of concept, it should be integrated in the compiler for an actual programming language, together with the implementation of the translation into E2.
- The modular application of the calculus calls for special attention.
- On the theoretical side, full descriptions should be published for the formal model and soundness proofs sketched in 4.12 and 6.3.
- The application to the frame problem must be clarified (in a companion article).
- Application to large programs requires both experimentation and theoretical analysis of the algorithms' complexity.

10 Acknowledgments

This article has benefited from discussions with Scott West, Stephan van Staden, Carlo Furia, Cristiano Calcagno, Yi Wei and Alexander Kogtenkov.

11 References

This work was made possible by the literature on software verification, particularly axiomatic semantics, separation logic, shape analysis, ownership, dynamic frames and static analysis. Two further references provide complementary background:

- [1] Bedřich Smetana: *Prodaná Nevěsta (The Bartered Bride)*, starring Gabriela Beňačková and Peter Dvorský, Supraphon, 1981, released as a DVD in 2006.
- [2] Jacques Offenbach (libretto by Meilhac and Halévy): *La Belle Hélène*, starring Felicity Lott, Michel Sénéchal, Laurent Naouri and Yann Beuron, conducted by Marc Minkowsky, 2000 (Théâtre du Châtelet), released as a DVD by Kultur Video in 2004.