Expression-based aliasing for OO–languages

Georgiana Caltais[1]

Department of Computer Science, ETH Zürich, Switzerland

Abstract. Alias analysis has been an interesting research topic in verification and optimization of programs. The undecidability of determining whether two expressions in a program may reference to the same object is the main source of the challenges raised in alias analysis. In this paper we propose an extension of a previously introduced alias calculus based on program expressions, to the setting of unbounded program executions such as infinite loops and recursive calls. Moreover, we devise a corresponding executable specification in the $\mathbb{K}$-framework. An important property of our extension is that, in a non-concurrent setting, the corresponding alias expressions can be over-approximated in terms of a notion of regular expressions. This further enables us to show that the associated $\mathbb{K}$-machinery implements an algorithm that always stops and provides a sound over-approximation of the "may aliasing" information, where soundness stands for the lack of false negatives. As a case study, we analyze the integration and further applications of the alias calculus in SCOOP. The latter is an object-oriented programming model for concurrency, recently formalized in Maude; $\mathbb{K}$ definitions can be compiled into Maude for execution.

1 Introduction

A research direction of interest in Computer Science is the application of *alias analysis* in verification and optimization of programs. One of the challenges along this line of research has been the undecidability of determining whether two expressions in a program *may* reference the same object. A rich suite of approaches aiming at providing a satisfactory balance between scalability and precision has already been developed in this regard. Examples include: (i) intra-procedural frameworks [16,15] that handle isolated functions only, and their inter-procedural counterparts [15,22,11] that consider the interactions between function calls; (ii) type-based techniques [8]; (iii) flow-based techniques [4,6] that establish aliases depending on the control-flow information of a procedure; (iv) context-(in)sensitive approaches [9,29] that depend on whether the calling context of a function is taken into account or not; (v) field-(in)sensitive approaches [20,1] that depend on whether the individual fields of objects in a program are traced or not. More details on such classifications can be found in [25], for instance. For a comprehensive survey on alias analyses for object-oriented programs, corresponding issues and remaining open problems, we refer the interested reader to the works in [28,10].

Of particular interest for the work in this paper is the untyped, flow-sensitive, field sensitive, inter-procedural and context-sensitive calculus for *may aliasing*, introduced in [14]. The aforementioned calculus covers most of the aspects of a modern object-oriented language, namely: object creation and deletion, conditionals, assignments, loops and (possibly recursive) function calls. The approach in [14] abstracts the aliasing information in terms of explicit access paths [17] referred to as *alias expressions*. Consider, for an example, the code

$$x := y;$$
$$\textbf{loop } x := x.next \textbf{ end} \tag{1}$$

The corresponding execution causes x to become aliased to $y.next.next. \ldots$, with a possibly infinite number of occurrences of the field *next*. The set of associated alias expressions can be equivalently written as:

$$\{[x, \, y.next^k] \mid k \geq 0\}. \tag{2}$$

The sources of imprecision introduced by the calculus in [14] are limited to ignoring tests in conditionals, and to "cutting at length L", for the case of possibly infinite alias relation as in (2). Intuitively, the cutting technique considers sequences longer than a given length L as aliased to all expressions.

There is a huge literature on heap analysis for aliasing [10], but hardly any paper that presents a calculus as in [14] allowing the derivation of alias relations as the result of applying various instructions of a programming language.

Our focus is two folded. First, we want extend the framework in [14] to the setting of unbounded program executions such as infinite loops and recursive calls. In accordance, the goal is to provide a way to shift from "finite" to "infinite behaviours". This can be achieved in a rather straightforward manner, by redefining the construct **loop** p **end** in [14] according to the informal semantics: "execute p repeatedly any number of times, including zero". However, developing a corresponding mechanism for reasoning on "may aliasing" in a finite number of steps is not trivial. The key observation that paves the way to a possible (finite state-based) modeling in a non-concurrent setting is that the alias expressions corresponding to loops and recursive calls grow in a regular fashion. Hence, they are finitely representable, as it is easy to see in (2), for instance. Such regularities cannot be exploited in concurrent contexts, due to the "non-determinism" of process interaction.

A similar technique exploiting regular behaviour of (non-concurrent) programs, in order to reason on "may aliasing", was previously introduced in [2]. In short, the results in [2] utilize abstract representations of programs in terms of finite pushdown systems, for which infinite execution paths have a regular structure (or are "lasso shaped") [3]. Then, in the style of abstract interpretation [7], the collecting semantics is applied over the (finite state) pushdown systems to obtain the alias analysis itself. In short, the main difference with the results in [2] consists in how the abstract memory addresses corresponding to pointer variables are represented. In [2] these range over a finite set of natural numbers. In this paper we consider alias expressions build according to the calculus in [14].

The work in [2] also proposes an implementation of pushdown systems in the $\mathbb{K}$-framework [26]. The latter is an executable semantic framework based on Rewriting Logic (RL) [18], and has successfully been used for defining programming languages and corresponding formal analysis tools. Moreover, $\mathbb{K}$ definitions have a direct implementation in $\mathbb{K}$-Maude [27].

We agree that it could be worth presenting our analysis as an abstract interpretation (AI) [7]. A modelling exploiting the machinery of AI (based on abstract domains, abstraction and concretization functions, Galois connections, fixed-points, *etc.*) is an interesting, but different research topic per se.

Our second interest w.r.t. may aliasing is its integration in SCOOP [21] – a simple object oriented programming model for concurrency; thus an operational based approach on handling the alias calculus is more appropriate. The basis of a RL-based framework for the design and analysis of the SCOOP model was recently set in [21]. The reference implementation of SCOOP is Eiffel [19]. The integration of alias analysis belongs to a more ambitious goal, namely, the construction of a RL-based toolbox for the analysis of SCOOP programs (examples include a deadlock detector and a type checker).

Our contribution. By drawing inspiration from, and building on top of the results in [14,2], in this paper we propose:

- an extension of the (finite) alias calculus in [14] to the setting of unbounded program executions, and a sound over-approximation technique based on "regular alias expressions", for non-concurrent settings;
- a RL-based specification of the extended calculus;
- an algorithm that always terminates and provides a sound over-approximation of "may aliasing" by exploiting a notion of regular (finitely representable) aliases, for non-concurrent settings.

Moreover, we analyze the integration, implementation and further applications of the alias calculus in SCOOP.

Paper structure. The paper is organized as follows. In Section 2 we introduce the extension of the alias calculus in [14] to unbounded executions. In Section 3 we provide the RL-based executable specification of the calculus in the $\mathbb{K}$ semantic framework. The implementation in SCOOP, and further applications are discussed in Section 4. In Section 5 we draw the conclusions and provide pointers to future work.

2 The alias calculus

In this section we define an extension of the calculus in [14], to unbounded program executions. Moreover, based on the idea behind the *pumping lemma for regular languages* [24], we devise a corresponding sound over-approximation of "may aliasing" in terms of regular expressions, applicable in sequential contexts. This paves the way to developing an algorithm for the aliasing problem, as presented in Section 3, in the formal setting of the $\mathbb{K}$ semantic framework [26].

Preliminaries. We proceed by briefly recalling the notion of *alias relation* and a series of associated notations and basic operations, as introduced in [14].

We call an *expression* a (possibly infinite) path of shape $x.y.z. \ldots$, where x is a local variable, class attribute or *Current*, and $y, z, \ldots$ are attributes. Here, *Current*, also known as *this* or *self*, stands for the current object. For an arbitrary alias expression e, it holds that $e. Current = Current.e = e$. Let E represent the set of all expressions of a program. An *alias relation* is a symmetric and irreflexive binary relation over $E \times E$.

Given an alias relation r and an expression e, we define

$$r/e = \{e\} \cup \{x \colon E \mid [x, e] \in r\}$$

denoting the set consisting of all elements in r which are aliased to e, plus e itself.

Let x be an expression; we write $r - x$ to represent r without the pairs with one element of shape $x.e$.

We say that an alias relation is *dot complete* whenever for any t, u, v and a it holds that if $[t, u]$ and $[t.a, v]$ are alias pairs, then $[u.a, v]$ is an alias pair and, moreover, if a is in the domain of t, then $[t.a, u.a]$ is an alias pair. By the "domain of t" we refer to a method or a field in the class corresponding to the object referred by the expression associated to t. For instance, given a class NODE with a field *next* of type NODE, and a NODE object x, we say that *next* is in the domain of $t = x.next.next$. For the sake of brevity, we write *dot-complete*(r) for the closure under dot-completeness of a relation r.

The notation $r[x = u]$ represents the relation r augmented with pairs $[x, y]$ and made dot complete, where y is an element of u.

2.1 Extension to unbounded executions

We further introduce an extension of the alias calculus in [14] to infinite alias relations corresponding to unbounded executions such as infinite loops or recursive calls. The main difference in our approach is reflected by the definition of loops, which now complies to the usual fixed-point denotational semantics.

The alias calculus is defined by a set of axioms "describing" how the execution a program affects the aliasing between expressions. As in [14], the calculus ignores tests in conditionals and loops. The *program instructions* are defined as follows:

$$p :: = p \,;\, p \mid \textbf{then } p \textbf{ else } p \textbf{ end} \mid \\ \textbf{create } x \mid \textbf{forget } x \mid t := s \mid \\ \textbf{loop } p \textbf{ end} \mid \textbf{call } f(l) \mid x.\textbf{call } f(l). \tag{3}$$

In short, we write $r \gg p$ to represent the alias information obtained by executing p when starting with the initial alias relation r.

The axiom for sequential composition is defined in the obvious way:

$$r \gg (p \,;\, q) = (r \gg p) \gg q. \tag{4}$$

Conditionals are handled by considering the union of the alias pairs resulted from the execution of the instructions corresponding to each of the two branches, when starting with the same initial relation:

$$r \gg (\textbf{then } p \textbf{ else } q \textbf{ end}) = r \gg p \ \cup \ r \gg q. \tag{5}$$

As previously mentioned, we define $r \gg \textbf{loop } p \textbf{ end}$ according to its informal semantics : "execute p repeatedly any number of times, including zero". The corresponding rule is:

$$r \gg (\textbf{loop } p \textbf{ end}) = \bigcup_{n \in \mathbb{N}} (r \gg p^n) \tag{6}$$

where $\cup$ stands for the union of alias relations, as above. This way, our calculus is extended to infinite alias relations. This is the main difference with the approach in [14] that proposes a "cutting" technique restricting the model to a maximum length L. In [14], sequences longer than L are considered as aliased to all expressions. Orthogonally, for sequential settings, we provide finite representations of infinite alias relations based on over-approximating regular expressions, as we shall see in Section 2.2.

Both the creation and the deletion of an object x eliminate from the current alias relation all the pairs having one element prefixed by x:

$$\begin{aligned} r \gg (\textbf{create } x) &= r - x \\ r \gg (\textbf{forget } x) &= r - x. \end{aligned} \tag{7}$$

The (qualified) function calls comply to their initial definitions in [14]:

$$\begin{aligned} r \gg (\textbf{call } f(l)) &= (r[f^\bullet{:}l]) \gg \ | \ f \ | \\ r \gg (x.\textbf{call } f(l)) &= x.((x'.r) \gg \textbf{call } f(x'.l)). \end{aligned} \tag{8}$$

Here $f^\bullet$ and $| \ f \ |$ stand for the formal argument list and the body of f, respectively, whereas $r[u{:}v]$ is the relation r in which every element of the list v is replaced by its counterpart in u. Intuitively, the negative variable x' is meant to transpose the context of the qualified call to the context of the caller. Note that "." (i.e., the constructor for alias expressions) is generalized to distribute over lists and relations: $x.[a, b, \ldots] = [x.a, x.b, \ldots]$.

For an example, consider a class C in an OO-language, and an associated procedure f that assigns a local variable y, defined as: $f(x) \{ \ y := \ x \ \}$. Then, for instance, the aliasing for $a.\textbf{call } f(a)$ computes as follows:

$$\begin{aligned} \emptyset \ \gg \ a.\textbf{call } f(a) &= \\ a.(a'.\emptyset \ \gg \ y := a'.a) &= \\ a.(\emptyset \ \gg \ y := Current) &= \\ dot\text{-}complete(\{[a.y, a]\}). \end{aligned}$$

Recursive function calls can lead to infinite alias relations. In sequential settings, as for the case of loops, the mechanism exploiting sound regular over-approximations in order to derive finite representations of such relations is presented in the subsequent sections.

The axiom for assignment is as well in accordance with its original counterpart in [14]:

$$r \gg (t := s) = \textbf{given } r_1 = r[ot = t]$$
$$\textbf{then } (r_1 - t)[t = (r_1/s - t)] - ot \textbf{ end} \qquad (9)$$

where ot is a fresh variable (that stands for "old t"). Intuitively, the aliasing information w.r.t. the initial value of t is "saved" by associating t and ot in r and closing the new relation under dot-completeness, in r_1. Then, the initial t is "forgotten" by computing $r_1 - t$ and the new aliasing information is added in a consistent way. Namely, we add all pairs (t, s'), where s' ranges over $r_1/s - t$ representing all expressions already aliased with s in r_1, including s itself, but without t. Recall that alias relations are not reflexive, thus by eliminating t we make sure we do not include pairs of shape $[t, t]$. Then, we consider again the closure under dot-completeness and forget the aliasing information w.r.t. the initial value of t, by removing ot.

Remark 1. It is worth discussing the reason behind *not* considering transitive alias relations. Assume the following program:

$$\textbf{then } x := y \textbf{ else } y := z \textbf{ end}$$

Based on the equations (5) and (9) handling conditionals and assignments, respectively, the calculus correctly identifies the alias set: $\{[x, y], [y, z]\}$. Including $[x, z]$ would be semantically equivalent to the execution of the two branches in the conditional at the same time, which is not what we want.

2.2 A sound over-approximation

In a sequential setting, the challenge of computing the alias information in the context of (infinite) loops and recursive calls reduces to evaluating their corresponding "unfoldings", captured by expressions of shape

$$r \gg p^\omega,$$

with ω ranging over naturals plus infinity, r an (initial) alias relation ($r = \emptyset$), and p a *basic control block* defined by:

$$p ::= p \, ; p \mid \textbf{then } p \textbf{ else } p \textbf{ end} \mid$$
$$\textbf{create } x \mid \textbf{forget } x \mid \qquad (10)$$
$$t := s.$$

The value $r \gg p^\omega$ refers to the alias relation obtained by recursively executing the control block p, and it is calculated in the expected way:

$$r \gg p^0 = r$$
$$r \gg p^{k+1} = (r \gg p^k) \gg p.$$

Consider again the code in (1):

$$x := y;$$
$$\textbf{loop } x := x.next \textbf{ end}.$$

Its execution generates the alias relation

$$(((\emptyset \gg (x := y)) \gg (x := x.next)) \gg (x := x.next) \dots$$

including an infinite number of pairs of shape:

$$[x, y.next], \ [x, y.next.next], \ [x, y.next.next.next] \dots \ . \tag{11}$$

A similar reasoning does not hold for concurrent applications, where process interaction is not "regular".

In what follows we provide a way to compute finite representations of infinite alias relations in sequential settings. The key observation is that alias expressions corresponding to unbounded program executions grow in a regular fashion. See, for instance, the aliases in (11), which are pairs of type $[x, y.next^{k \geq 1}]$.

Regular expressions are defined similarly to the regular languages over an alphabet. We say that an expression is *regular* if it is a local variable, class attribute or *Current*. Moreover, the concatenation $e_1 . e_2$ of two regular expressions e_1 and e_2 is also regular. Given a regular alias expression e, the expression e^* is also regular; here $(-)^*$ denotes the Kleene star [13]. We call an alias relation *regular* if it consists of pairs of regular expressions.

Lemma 1. *Assume p a program built according to the rules in (3). Then, in a sequential setting, the relation $\emptyset \gg p$ is regular.*

Proof. The result follows by induction on the structure of p. We refer to Appendix A for the detailed proof.

Inspired by the idea behind the *pumping lemma for regular languages* [24], we define a *lasso* property for alias relations, which identifies the repetitive patterns within the structure of the corresponding alias expressions. The intuition is that such patterns will occur for an infinite number of times due to the execution of loops or recursive function calls. Then, we supply sound over-approximations of "lasso" relations, based on regular alias expressions.

In the context of alias relations, we say that the lasso property is satisfied by r and r' whenever the following two conditions hold: (1) r behaves like a *lasso base* of r'. Namely, all the pairs $[e_1, e_2] \in r$ are used to generate elements $[e'_1, e'_2] \in r'$, by repeating tails of prefixes of e_1 and e_2, respectively, and (2) r' is a *lasso extension* of r. Namely, all the pairs in r' are generated from elements of r by repeating tails of their prefixes. For example, if e_1 above is an expression of shape $x.y.z.w$, then e'_1 can be $x.y.y.z.w$ if we consider the tail y of the prefix $x.y$, or $x.y.z.y.z.w$ if we take the tail $y.z$ of the prefix $x.y.z$.

Formally, consider r and r' two alias relations, and x_i, y_i and z_i a set of (possibly empty) expressions, for $i \in \{1, 2\}$. Then:

$$\text{lasso}(r, r') = ([x_1 y_1 z_1, x_2 y_2 z_2] \in r \ \text{ iff } \ [x_1 y_1 y_1 z_1, x_2 y_2 y_2 z_2] \in r'). \tag{12}$$

For the simplicity of notation we sometimes omit the dot-separators between expressions. For instance, we write $x\,y\,z$ in lieu of $x.y.z$.

Assuming a lasso over r and r', we compute a relation consisting of regular expressions over-approximating r and r' as:

$$\mathrm{reg}(r, r') = \{[x_1 y_1^* z_1, x_2 y_2^* z_2] \mid \\ [x_1 y_1 z_1, x_2 y_2 z_2] \in r \wedge \\ [x_1 y_1 y_1 z_1, x_2 y_2 y_2 z_2] \in r'\} \tag{13}$$

where x_i, y_i and z_i are possibly empty expressions, for $i \in \{1, 2\}$. As previously indicated, the over-approximation is sound w.r.t. the repeated application of a basic control block as in (10), in the way that it does not introduce any false negatives:

Lemma 2. *Consider r and r' two alias relations, and p a basic control block in a sequential setting. If $r \gg p = r'$ and $\mathrm{lasso}(r, r') = true$, then the following holds for all $n \geq 1$:*
$$r \gg p^n \in \mathrm{reg}(r, r').$$

Proof. The reasoning is by induction on n. The base case follows immediately, whereas the induction step is proved by "reductio ad absurdum". A detailed proof is included in Appendix B. □

3   A $\mathbb{K}$-machinery for collecting aliases

In this section we provide the specification of a RL-based mechanism collecting the alias information in the $\mathbb{K}$ semantic framework [26]. We choose $\mathbb{K}$ more as a notational convention to enable compact and modular definitions. In reality, the $\mathbb{K}$-rules in this section are implemented in Maude, as rewriting theories, on top of the formalization of SCOOP [21] (we refer to Section 4 for more details on our approach).

In short, our strategy is to start with a program built on top of the control structures in (3), then to apply the corresponding $\mathbb{K}$-rules in order to get the "may aliasing" information in a designated $\mathbb{K}$-cell ($\langle\,-\,\rangle_{\mathrm{al}}$). Independently of the setting (sequential or concurrent) one can exploit this approach in order to evaluate the aliases of a given finite length L. We also show that for sequential contexts, the application of the $\mathbb{K}$-rules is finite and the aliases in the final configuration soundly over-approximate the (infinite) "may alias" relations of the calculus.

Brief overview of $\mathbb{K}$. $\mathbb{K}$ [26] is an executable semantic framework based on Rewriting Logic [18]. It is suitable for defining (concurrent) languages and corresponding formal analysis tools, with straightforward implementation in $\mathbb{K}$-Maude [27]. $\mathbb{K}$-definitions make use of the so-called *cells*, which are labelled and can be nested, and (rewriting) *rules* describing the intended (operational) semantics.

A *cell* is denoted by $\langle - \rangle_{[name]}$, where [name] stands for the *name of the cell*. A construction $\langle\ .\ \rangle_n$ stands for an *empty cell* named n. We use "pattern matching" and write $\langle\ c\ \dots\rangle_n$ for a cell with content c at the top, followed by an arbitrary content $(\dots)$. Orthogonally, we can utilize cells of shape $\langle\dots\ c\ \rangle_n$ and $\langle\ \dots c\dots\ \rangle_n$, defined in the obvious way.

Of particular interest is $\langle\ -\ \rangle_k$ – the *continuation cell*, or the *k-cell*, holding the stack of program instructions (associated to one processor), in the context of a programming language formalization. We write

$$\langle\ i_1 \curvearrowright i_2\ \dots\rangle_k$$

for a set of instructions to be "executed", starting with instruction i_1, followed by i_2. The associative operation $\curvearrowright$ is the instruction sequencing.

A $\mathbb{K}$-rewrite rule

$$\langle\ c\ \dots\rangle_{n_1}\langle\ c'\ \rangle_{n_2}\ \Rightarrow\ \langle\ c'\ \dots\rangle_{n_1}\langle\dots\ c'\ \rangle_{n_3} \tag{14}$$

reads as: if cell n_1 has c at the top and cell n_2 contains value c', then c is replaced by c' in n_1 and c' is added at the end of the cell n_3. The content of n_2 remains unchanged. In short, (14) is written in a $\mathbb{K}$-like syntax as:

$$\langle\ \frac{c}{c'}\ \dots\rangle_{n_1}\ \langle\ c'\ \rangle_{n_2}\ \langle\dots\ \frac{.}{c'}\ \rangle_{n_3}\ .$$

We further provide the details behind the $\mathbb{K}$-specification of the alias calculus. As expected, the k-cell retains the instruction stack of the object-oriented program. We utilize cells $\langle-\rangle_{al}$ to enclose the current alias information, and the so-called *back-tracking cells* $\langle-\rangle_{bkt-\dots}$ enabling the sound computation of aliases for the case of $\textbf{then} - \textbf{else} - \textbf{end}$ and, in non-concurrent contexts, for loops and (possibly recursive) function calls. As a convention, we mark with ($\clubsuit$) the rules that are sound only for non-concurrent applications, based on Lemma 2. Due to space limitations, in what follows we introduce only the $\mathbb{K}$-rules for handling assignments and loops. The entire specification is included in Appendix C.

As expected, the assignment rule simply restores the current alias relation according to its axiom in (9), and removes the assignment instruction from the top of the k-cell:

$$\langle\ \frac{r}{(r_1 - t)[t = (r_1/s\ -\ t)] - ot}\ \rangle_{al}\ \langle\ \frac{t := s}{.}\ \dots\rangle_k \quad \text{with } r_1 = r[ot = t] \tag{15}$$

For $\textbf{loop } p\ \textbf{end}$, we utilize a meta-construction $p\ \boxed{1}\ \textbf{loop } p\ \textbf{end}$ simulating the unfolding corresponding to (6), and a back-tracking stack $\langle-\rangle_{bkt-l}$ collecting the alias information obtained after each execution of p. Moreover, the $\mathbb{K}$-implementation exploits the result in Lemma 2. Whenever a "lasso" is reached, the infinite rewriting is prevented by resuming the infinite application of p in terms of a sound over-approximating alias relation. The $\mathbb{K}$-rules are as follows.

First, the aforementioned unfolding is performed, and the alias relation before p is stored in the back-tracking cell as $\langle r\rangle_{al-o}\langle p\rangle_l$:

$$\langle\, r\,\rangle_{\text{al}}\, \left\langle\, \frac{\textbf{loop } p \textbf{ end}}{p\,\boxed{1}\,\textbf{loop } p \textbf{ end}}\, \ldots\right\rangle_{\text{k}}\, \left\langle\, \frac{\cdot}{\langle\, r\,\rangle_{\text{al-o}}\langle\, p\,\rangle_{\text{l}}}\, \ldots\right\rangle_{\text{bkt-l}} \tag{16}$$

If the alias relation r' obtained after the successful execution of p (marked by $\boxed{1}$ at the top of the continuation) is not a lasso of the aliasing r before p (previously stored in $\langle-\rangle_{\text{bkt-l}}$) then p is constrained to a new execution by becoming the top of the k-cell, and r' is memorized for back-tracking:

$$\langle\, r'\,\rangle_{\text{al}}\, \left\langle\, \frac{\boxed{1}\,\textbf{loop } p \textbf{ end}}{p\,\boxed{1}\,\textbf{loop } p \textbf{ end}}\, \ldots\right\rangle_{\text{k}}\, \left\langle\, \frac{\langle\, r\,\rangle_{\text{al-o}}\langle\, p\,\rangle_{\text{l}}}{\langle\, r'\,\rangle_{\text{al-o}}\langle\, p\,\rangle_{\text{l}}}\, \ldots\right\rangle_{\text{bkt-l}} \text{ if not lasso}(r,r') \ (\clubsuit) \tag{17}$$

Last, if a lasso is reached after the execution of p, then the current aliasing is soundly replaced by a "regular" over-approximation $\text{reg}(r,r')$, the corresponding back-tracking information is removed from $\langle-\rangle_{\text{bkt-l}}$ and the **loop** instruction is eliminated from the k-cell:

$$\left\langle\, \frac{r'}{\text{reg}(r,r')}\, \right\rangle_{\text{al}}\left\langle\, \frac{\boxed{1}\,\textbf{loop } p \textbf{ end}}{\cdot}\, \ldots\right\rangle_{\text{k}}\left\langle\, \frac{\langle\, r\,\rangle_{\text{al-o}}\langle\, p\,\rangle_{\text{l}}}{\cdot}\, \ldots\right\rangle_{\text{bkt-l}} \text{ if lasso}(r,r') \ (\clubsuit) \tag{18}$$

In a non-concurrent setting, the machinery orchestrating the $\mathbb{K}$-rules introduced in this section, and thoroughly discussed in Appendix C, implements an algorithm that always terminates and provides a sound over-approximation of "may aliasing".

Theorem 1. *Consider p a program built on top of the control structures in (3), that executes in a sequential setting. Then, the application of the corresponding $\mathbb{K}$-rules when starting with p and an empty alias relation, is a finite rewriting of shape*

$$\langle\, \emptyset\,\rangle_{\text{al}}\langle\, p\,\rangle_{\text{k}}\, \overset{(*)}{\Longrightarrow}\, \langle\, r\,\rangle_{\text{al}}\langle\, \cdot\,\rangle_{\text{k}},$$

with r a sound over-approximation of the aliasing information corresponding to the execution of p.

Proof. The key observation is that, due to the execution of loops and/or recursive calls, expressions can infinitely grow in a *regular* fashion. Hence, a lasso is always reached. Consequently, the control structure generating the infinite behaviour is removed from the k-cell, according to the associated $\mathbb{K}$-specification for loops and/or recursive calls. This guarantees termination. Moreover, recall that the regular expressions replacing the current alias information are a sound over-approximation, according to Lemma 2. $\qquad\square$

Observe that the *RL*-based machinery can simulate precisely the "cutting at length L" technique in [14]. It suffices to disable the rules ($\clubsuit$) and stop the rewriting after L steps.

The naturalness of applying the resulted aliasing framework is illustrated in the example in Appendix D, for the case of two mutually recursive functions.

4   Integration in SCOOP

In this section we provide a brief overview on the integration and applicability of the alias calculus in SCOOP [21] – a simple object-oriented programming model for concurrency. Two main characteristics make SCOOP simple: 1) just one keyword programmers have to learn and use in order to enable concurrent executions, namely, *separate* and 2) the burden of orchestrating concurrent executions is handled within the model, therefore reducing the risk of correctness issues.

In short, the key idea of SCOOP is to associate to each object a processor, or *handler* (that can be a CPU, or it can also be implemented in software, as a process or thread). Assume a processor p that performs a call $o.f()$ on an object o. If o is declared as "separate", then p sends a request for executing $f()$ to q – the handler of o (note that p and q can coincide). Meanwhile, p can continue. Processors communicate via *channels*.

The Maude semantics of SCOOP in [21] is defined over tuples of shape

$$\langle p_1 :: St_1 \mid \ldots \mid p_n :: St_n, \sigma \rangle$$

where, p_i denotes a processor (for $i \in \{1, \ldots, n\}$), St_i is the call stack of p_i and σ is the *state* of the system. States hold the information about the *heap* (which is a mapping of references to objects) and the *store* (which includes formal arguments, local variables, *etc.*).

The assignment instruction, for instance, is formally specified as the transition rule:

$$\frac{\text{a is fresh}}{\Gamma \vdash \langle p :: t := s; St, \sigma \rangle \rightarrow \langle p :: \text{eval}(a, s); \text{wait}(a); \text{write}(t, a.data); St, \sigma \rangle} \quad (19)$$

where, intuitively, "eval(a, s)" evaluates s and puts the result on channel a, "wait(a)" enables processor p to use the evaluation result, "write$(t, a.data)$" sets the value of t to $a.data$, St is a call stack, and Γ is a typing environment [23] containing the class hierarchy of a program and all the type definitions.

At this point it is easy to understand that the 𝕂-rule for assignments

$$\langle \frac{r}{(r_1 - t)[t = (r_1/s - t)] - ot} \rangle_{\text{al}} \langle \frac{t := s \ \ldots}{.} \rangle_{\text{k}} \quad \text{with } r_1 = r[ot = t] \quad (15)$$

can be straightforwardly integrated in (19) by enriching the state structure with a new field encapsulating the alias information, and considering instead the transition $\Gamma \vdash \langle p :: t := s; St, \sigma \rangle \rightarrow \langle p :: \text{eval}(a, s); \text{wait}(a); \text{write}(t, a.data); St, \sigma' \rangle$ where

$$\sigma.aliases = r \qquad \sigma'.aliases = (r_1 - t)[t = (r_1/s - t)] - ot$$

with r and r_1 as in (15). The integration of all the 𝕂-rules of the alias calculus on top of the Maude formalization of SCOOP can be achieved by following a similar approach.

For a case study, one can download the SCOOP formalization at:
`https://dl.dropboxusercontent.com/u/1356725/SCOOP.zip`
and run the command
`> maude SCOOP.maude ..\examples\aliasing-linked_list.maude`
corresponding to the code in (1):

$$x := y; \ \mathbf{loop} \ x := x.next \ \mathbf{end}.$$

The console outputs the aliased expressions for a rewriting of depth 100 which include, as expected, pairs of shape $[x, y.next^k]$. (The over-approximating mechanism for sequential settings is still to be implemented.)

As can be observed based on the code in `aliasing-linked_list.maude`, in order to implement our applications in Maude, we use intermediate (still intuitive) representations. For instance, the class structure defining a node in a simple linked list, with filed *next* is declared as:

```
class 'NODE
    create {'make}
    ( attribute { 'ANY } 'next : [?, . , 'NODE] ; )
    [...]
end ;
```

where `'next : [?, . , 'NODE]` stands for an object of type NODE, that is handled by the current processor (.) and that can be Void (?), and `'make` plays the role of a constructor. The intermediate representation of the instruction block in (1) is:

```
assign ('x, 'y);
until False loop ( assign ('x, 'x . 'next(nil)) ; ) end ;
```

We include in Appendix E the whole class structure corresponding to (1), together with (the relevant parts of) the console output. For a detailed description of SCOOP and its Maude formalization we refer the interested reader to the work in [21].

4.1 Further applications of the alias calculus

Apart from providing an alias analysis tool, the alias calculus can be exploited in order to build an abstract semantics of SCOOP. For example, an abstraction of the assignment rule (15) would omit the evaluation of the right-hand side of the assignment $t := s$ and the associated message passing between channels:

$$\frac{\cdot}{\Gamma \vdash \langle p :: t := s; St, \sigma \rangle \to \langle p :: St, \sigma' \rangle}$$

where

$$\sigma.aliases = r \qquad \sigma'.aliases = (r_1 - t)[t = (r_1/s - t)] - ot$$

with r and r_1 as in (15). This way one derives a simplified, reduced semantics of SCOOP, more appropriate for model checking, for instance; the current

SCOOP formalization in Maude is often too large for this purpose. A survey on abstracting techniques on top of Maude executable semantics is provided in [18].

Furthermore, the aliasing information could be used for the so-called "deadlocking" problem, where two or more executing threads are each waiting for the other to finish. In the context of SCOOP, this is equivalent to identifying whether a set of processors reserve each other circularly (*i.e.*, there is a Coffman deadlock). This situation might occur, for instance, in a Dinning Philosophers scenario, where both philosophers and forks are objects residing on their own processors. The difficulty of identifying such deadlocks stems from the fact that SCOOP processors are known from object references, which *may be aliased*.

5   Conclusions

In this paper we provide an extension of the alias calculus in [14] from finite alias relations to infinite ones corresponding to loops and recursive calls. Moreover, we devise an associated executable specification in the $\mathbb{K}$ semantic framework [26]. In Theorem 1 we show that the RL-based machinery implements an algorithm that always terminates with a sound over-approximation of "may aliasing", in non-concurrent settings. This is achieved based on the sound (finitely representable) over-approximation of ("lasso shaped") alias expressions in terms of regular expressions, as in Lemma 2. We also discuss the integration and applicability of the alias calculus on top of the Maude formalization of SCOOP [21].

An immediate direction for future work is to identify interesting (industrial) case studies to be analyzed using the framework developed in this paper. We are also interested in devising heuristics comparing the efficiency and the precision (*e.g.*, the number of false positives introduced by the alias approximations) between our approach and other aliasing techniques. In this respect, we anticipate that the rewriting modulo associativity, together with the pattern matching capabilities of Maude will accelerate the identification of the "lasso" properties and the corresponding over-approximating regular alias expressions. This could eventually provide an effective reasoning apparatus for the "may aliasing" problem.

Another research direction is to derive alias-based abstractions for analyzing concurrent programs. We foresee possible connections with the work in [12] on *concurrent Kleene algebra* formalizing choice, iteration, sequential and concurrent composition of programs. The corresponding definitions exploit abstractions of programs in terms of traces of events that can depend on each other. Thus, obvious challenges in this respect include: (i) defining notions of dependence for all the program constructs in this paper, (ii) relating the concurrent Kleene operators to the semantics of the SCOOP concurrency model and (iii) checking whether fixed-points approximating the aliasing information can be identified via fixed-point theorems.

Furthermore, it would be worth investigating whether the graph-based model of alias relations introduced in [14] can be exploited in order to derive finite $\mathbb{K}$ specifications of the extended alias calculus. In case of a positive answer, the

general aim is to study whether this type of representation increases the speed of the reasoning mechanism, and why not – its accuracy. With the same purpose, we refer to a possible integration with the technique in [5] that handles point-to graphs via a stack-based algorithm for fixed-point computations.

We are also interested to what extent an abstract semantics based on aliases for SCOOP can be exploited for building more efficient analysis tools such as deadlock detectors, for instance. A survey on similar techniques that abstract away from possibly irrelevant information w.r.t. the problem under consideration is provided in [18].

References

1. E. Albert, P. Arenas, S. Genaim, and G. Puebla. Field-sensitive value analysis by field-insensitive analysis. In *Proceedings of the 2Nd World Congress on Formal Methods*, FM '09, pages 370–386, Berlin, Heidelberg, 2009. Springer-Verlag.
2. I. M. Asavoae. Abstract semantics for alias analysis in K. *Electr. Notes Theor. Comput. Sci.*, 304:97–110, 2014.
3. A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *CONCUR*, pages 135–150, 1997.
4. M. Burke, P. Carini, J.-D. Choi, and M. Hind. Flow-insensitive interprocedural alias analysis in the presence of pointers. In K. Pingali, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, volume 892 of *Lecture Notes in Computer Science*, pages 234–250. Springer Berlin Heidelberg, 1995.
5. D. R. Chase, M. N. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *PLDI*, pages 296–310, 1990.
6. J.-D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, pages 232–245, New York, NY, USA, 1993. ACM.
7. P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *J. Log. Program.*, 13(2&3):103–179, 1992.
8. A. Diwan, K. S. McKinley, and J. E. B. Moss. Type-based alias analysis. *SIGPLAN Not.*, 33(5):106–117, May 1998.
9. M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 242–256, New York, NY, USA, 1994. ACM.
10. M. Hind. Pointer analysis: haven't we solved this problem yet? In *PASTE*, pages 54–61, 2001.
11. M. Hind, M. Burke, P. Carini, and J.-D. Choi. Interprocedural pointer alias analysis. *ACM Trans. Program. Lang. Syst.*, 21(4):848–894, July 1999.

12. C. A. R. Hoare, B. Möller, G. Struth, and I. Wehrman. Concurrent Kleene algebra. In *CONCUR 2009 - Concurrency Theory, 20th International Conference, CONCUR 2009, Bologna, Italy, September 1-4, 2009. Proceedings*, pages 399–414, 2009.

13. S. C. Kleene. Representation of events in nerve nets and finite automata. In C. Shannon and J. McCarthy, editors, *Automata Studies*, pages 3–41. Princeton University Press, Princeton, NJ, 1956.

14. A. Kogtenkov, B. Meyer, and S. Velder. Alias and change calculi, applied to frame inference. *CoRR*, abs/1307.3189, 2013.

15. W. Landi. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst.*, 1(4):323–337, Dec. 1992.

16. W. Landi and B. G. Ryder. Pointer-induced aliasing: A problem classification. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '91, pages 93–103, New York, NY, USA, 1991. ACM.

17. J. R. Larus and P. N. Hilfinger. Detecting conflicts between structure accesses. In *PLDI*, pages 21–34, 1988.

18. J. Meseguer and G. Rosu. The rewriting logic semantics project: A progress report. In *Fundamentals of Computation Theory - 18th International Symposium, FCT 2011, Oslo, Norway, August 22-25, 2011. Proceedings*, pages 1–37, 2011.

19. B. Meyer. *Eiffel: The Language*. Prentice-Hall, 1991.

20. A. Miné. Field-sensitive value analysis of embedded c programs with union types and pointer arithmetics. In *Proceedings of the 2006 ACM SIGPLAN/SIGBED Conference on Language, Compilers, and Tool Support for Embedded Systems*, LCTES '06, pages 54–63, New York, NY, USA, 2006. ACM.

21. B. Morandi, M. Schill, S. Nanz, and B. Meyer. Prototyping a concurrency model. In *ACSD*, pages 170–179, 2013.

22. E. M. Myers. A precise inter-procedural data flow algorithm. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '81, pages 219–230, New York, NY, USA, 1981. ACM.

23. P. Nienaltowski. *Practical Framework for Contract-based Concurrent Object-oriented Programming*. ETH, 2007.

24. M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM J. Res. Dev.*, 3(2):114–125, Apr. 1959.

25. V. Robert and X. Leroy. A formally-verified alias analysis. In *CPP*, pages 11–26, 2012.

26. G. Rosu and T. F. Serbanuta. K overview and SIMPLE case study. In *Proceedings of International K Workshop (K'11)*, ENTCS. Elsevier, 2013. To appear.

27. T.-F. Serbanuta and G. Rosu. K-Maude: A rewriting based tool for semantics of programming languages. In *WRLA*, pages 104–122, 2010.

28. M. Sridharan, S. Chandra, J. Dolby, S. J. Fink, and E. Yahav. Alias analysis for object-oriented programs. In *Aliasing in Object-Oriented Programming*, pages 196–232. 2013.

29. R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, PLDI '95, pages 1–12, New York, NY, USA, 1995. ACM.

A Regular expressions in sequential settings

In this section we provide the proof of Lemma 1; we proceed by demonstrating a series of intermediate results.

Remark 2. We first observe that the operations r/s, $r - x$, dot-completeness and $r[x = u]$ introduced in Section 2 preserve the regularity of an alias relation r.

Then, we define a notion of *finite execution* control blocks:

$$p :: = \textbf{create } x \mid \textbf{forget } x \mid t := s \mid$$
$$p\,;p \mid \textbf{then } p \textbf{ else } p \textbf{ end} \mid \qquad (20)$$
$$\textbf{call } f(l) \mid x.\textbf{call } f(l)$$

where f stands for a non-recursive function.

It is easy to see that the execution of control blocks as in (20) preserve the regularity of alias relations as well.

Lemma 3. *For all regular alias relations r and p a finite-execution control block, in a sequential setting, it holds that $r \gg p$ is also regular.*

Proof. The proof follows immediately, by induction on the structure of p and Remark 2. Base cases are: **create** x, **forget** x and $t := s$. For function calls, the result is a consequence of their corresponding unfolding, based on the definitions in (8).

Remark 3. W.r.t. may aliasing, recursive calls can be handled via loops. Consider, for instance the recursive function

$$f(x) \{ B_1;\ f(y);\ B_2 \}$$

where B_1 and B_2 are instruction blocks built as in (3). It is intuitive to see that computing the may aliases resulted from the execution of $f(x)$ reduces executing unfoldings of shape:

$$\textbf{loop } B_1 \textbf{ end};\ \textbf{loop } B_2 \textbf{ end}.$$

Moreover, unbounded program executions also preserve regularity.

Lemma 4. *For all regular alias relations r and p a control block that can execute unboundedly, in a sequential setting, it holds that $r \gg p$ is also regular.*

Proof. The proof follows by induction on the number of nested loops in p and Remark 3.

Then, the result in Lemma 1 follows immediately by Lemma 3 and Lemma 4.

B Sound over-approximations

In what follows we provide the proof of Lemma 2:
Consider r and r' two alias relations, and p a basic control block. If $r \gg p = r'$ and $\mathrm{lasso}(r, r') = true$, then the following holds for all $n \geq 1$:

$$r \gg p^n \in \mathrm{reg}(r, r').$$

Proof. We proceed by induction on n.

– *Base case*: $n = 1$. By hypothesis it holds that $\mathrm{lasso}(r, r') = true$. Hence, according to the definition of $\mathrm{lasso}(-, -)$ in (12), there exists a one-to-one correspondence of the shape

$$[x_1 y_1 z_1, x_2 y_2 z_2] \in r \quad \text{iff} \quad [x_1 y_1 y_1 z_1, x_2 y_2 y_2 z_2] \in r'$$

between the elements of r and r', respectively.
Consequently, by the definition of $\mathrm{reg}(-, -)$ in (13), it is easy to see that

$$r' \in \mathrm{reg}(r, r').$$

– *Induction step.* Fix a natural number n and suppose that

$$r \gg p^k \in \mathrm{reg}(r, r') \tag{21}$$

for all $k \in \{1, \ldots, n\}$. We want to prove that (21) holds also for $k = n + 1$. We continue by "reductio ad absurdum". Consider

$$\overline{r} = r \gg p^n \in \mathrm{reg}(r, r'),$$

and assume that

$$\overline{r} \gg p \notin \mathrm{reg}(r, r') \tag{22}$$

Clearly, the execution of p when starting with $\overline{r}$ identifies an alias pair which is not in $\mathrm{reg}(r, r')$. Given that p is a basic control block as in (10), and based on the corresponding definitions in (4)–(9), it is not difficult to observe that the regular structure of the alias information can only be broken via a new added pair (t, s') associated to an assignment $t := s$ within p.
Let $p = C[t := s]$, where C is a context built according to (10), and $t := s$ is the upper-most assignment instruction in the syntactic tree associated to p, that introduces a pair $[t, s']$ which is not in $\mathrm{reg}(r, r')$. Assume that $\tilde{r}$ is the intermediate alias relation obtained by reducing $\overline{r} \gg C[t := s]$ according to the equations (4)–(9), before the application of the assignment axiom corresponding to $t := s$.
Note that $t := s$ was executed at least once before, as $n \geq 1$, and observe that $\tilde{r} \in \mathrm{reg}(r, r')$. Hence, we identify two situations in the context of the aforementioned execution: (a) either all the newly added pairs corresponding

to the assignment $t := s$ complied to the regular structure, or (b) each new pair $[t', s']$ that did not fit the regular pattern was later removed via a subsequent instruction "**create** u" or "**forget** u" within p, with u a prefix of t' or s'.

If the case (a) above was satisfied, then, based on the definition of dot-completeness, a pair

$$(t, s') \in (\tilde{r_1} - t)[t = \tilde{r_1}/s - t] - ot,$$

where

$$\tilde{r_1} = \tilde{r}[ot = t]$$

cannot break the regular pattern of the alias expressions either. For the case (b) above, all the "non-well-behaved" new pairs will be again removed via a subsequent "**create** u" or "**forget** u" within p.

Therefore, the assumption in (22) is false, so it holds that:

$$\overline{r} \gg p = r \gg p^{n+1} \in \mathrm{reg}(r, r').$$

$\square$

C   Alias calculus in $\mathbb{K}$ − complete specification

In this section we provide the full specification of the alias calculus in $\mathbb{K}$. Recall that, as a convention, we mark with ($\clubsuit$) the rules that are sound only for non-concurrent contexts, based on Lemma 2.

The following $\mathbb{K}$-rules are straightforward, based on the axioms (4)–(9) in Section 2.1. Namely, the rule implementing an instruction $p \,;q$ simply forces the sequential execution of p and q by positioning $p \curvearrowright q$ at the top of the continuation cell:

$$\langle \; \frac{p \,;q}{p \curvearrowright q} \; \ldots \rangle_{\mathrm{k}} \tag{23}$$

Handling **create** x and **forget** x complies to the associated definitions. Namely, it updates the current alias relation by removing all the pairs having (at least) one element with x as prefix. In addition, it also pops the corresponding instruction from the continuation stack:

$$\langle \; \frac{r}{r - x} \; \rangle_{\mathrm{al}} \langle \; \frac{\mathbf{create}\ x}{.} \; \ldots \rangle_{\mathrm{k}} \qquad \langle \; \frac{r}{r - x} \; \rangle_{\mathrm{al}} \langle \; \frac{\mathbf{forget}\ x}{.} \; \ldots \rangle_{\mathrm{k}} \tag{24}$$

The assignment rule restores the current alias relation according to its axiom in (9), and removes the assignment instruction from the top of the k-cell:

$$\langle \; \frac{r}{(r_1 - t)[t = (r_1/s - t)] - ot} \; \rangle_{\mathrm{al}} \langle \; \frac{t := s}{.} \; \ldots \rangle_{\mathrm{k}} \quad \text{with } r_1 = r[ot = t] \tag{25}$$

The $\mathbb{K}$-implementation of a **then** p **else** q **end** statement is more sophisticated, as it instruments a stack-based mechanism enabling the computation of the union of alias relations $r \gg p \cup r \gg q$ in three steps. First, we define the $\mathbb{K}$-rule:

$$\langle\ r\ \rangle_{\mathrm{al}}\ \Big\langle\ \frac{\textbf{then } p \textbf{ else } q \textbf{ end}}{p\ \boxed{\text{et}}\ q\ \boxed{\text{ee}}}\ \ldots\Big\rangle_{\mathrm{k}}\ \Big\langle\ \frac{\cdot}{\langle\ r,p\ \rangle_{\mathrm{t}}\ \langle\ r,q\ \rangle_{\mathrm{e}}}\ \ldots\Big\rangle_{\mathrm{bkt\text{-}te}} \tag{26}$$

saving at the top of the back-tracking stack $\langle-\rangle_{\mathrm{bkt\text{-}te}}$ the initial alias relation r to be modified by both p and q, via two cells $\langle r,p\rangle_{\mathrm{t}}$ and $\langle r,q\rangle_{\mathrm{e}}$, respectively. Note that the original instruction in the k-cell is replaced by a meta-construction marking the end of the executions corresponding to the **then** and **else** branches with $\boxed{\text{et}}$ and $\boxed{\text{ee}}$, respectively.

Second, whenever the successful execution of p (signaled by $\boxed{\text{et}}$) at the top of the k-cell) builds an alias relation r', the execution of q starting with the original relation r is forced by replacing r' with r in $\langle-\rangle_{\mathrm{al}}$, and by positioning $q\ \boxed{\text{ee}}$ at the top of the k-cell. The new alias information after p, denoted by $\langle r',p\rangle_{\mathrm{t}}$, is updated in the back-tracking cell:

$$\Big\langle\ \frac{r'}{r}\ \Big\rangle_{\mathrm{al}}\ \Big\langle\ \frac{\boxed{\text{et}}\ q\ \boxed{\text{ee}}}{q\ \boxed{\text{ee}}}\ \ldots\Big\rangle_{\mathrm{k}}\ \Big\langle\ \frac{\langle\ r,p\ \rangle_{\mathrm{t}}}{\langle\ r',p\ \rangle_{\mathrm{t}}}\langle\ r,q\ \rangle_{\mathrm{e}}\ \ldots\Big\rangle_{\mathrm{bkt\text{-}te}} \tag{27}$$

Eventually, if the successful execution of q (marked by $\boxed{\text{ee}}$ at the top of $\langle-\rangle_{\mathrm{k}}$) produces an alias relation r'', then the final alias information becomes $r' \cup r''$, where r' is the aliasing after p, stored as showed in (27). The corresponding back-tracking information is removed from $\langle-\rangle_{\mathrm{bkt\text{-}te}}$, and the next program instruction is enabled in the k-cell:

$$\Big\langle\ \frac{r''}{r' \cup r''}\ \Big\rangle_{\mathrm{al}}\ \Big\langle\ \frac{\boxed{\text{ee}}}{\cdot}\ \ldots\Big\rangle_{\mathrm{k}}\ \Big\langle\ \frac{\langle\ r',p\ \rangle_{\mathrm{t}}\ \langle\ r,q\ \rangle_{\mathrm{e}}}{\cdot}\ \ldots\Big\rangle_{\mathrm{bkt\text{-}te}} \tag{28}$$

For **loop** p **end**, we utilize a meta-construction $p\ \boxed{\text{l}}\ \textbf{loop } p \textbf{ end}$ simulating the set union in (6), and a back-tracking stack $\langle-\rangle_{\mathrm{bkt\text{-}l}}$ collecting the alias information obtained after each execution of p. Moreover, the $\mathbb{K}$-implementation exploits the result in Lemma 2. Whenever a "lasso" is reached, the infinite rewriting is prevented by resuming the infinite application of p in terms of a sound over-approximating alias relation. The $\mathbb{K}$-rules are as follows.

First, the aforementioned unfolding is performed, and the alias relation before p is stored in the back-tracking cell as $\langle r\rangle_{\mathrm{al\text{-}o}}\langle p\rangle_{\mathrm{l}}$:

$$\langle\ r\ \rangle_{\mathrm{al}}\ \Big\langle\ \frac{\textbf{loop } p \textbf{ end}}{p\ \boxed{\text{l}}\ \textbf{loop } p \textbf{ end}}\ \ldots\Big\rangle_{\mathrm{k}}\ \Big\langle\ \frac{\cdot}{\langle\ r\ \rangle_{\mathrm{al\text{-}o}}\langle\ p\ \rangle_{\mathrm{l}}}\ \ldots\Big\rangle_{\mathrm{bkt\text{-}l}} \tag{29}$$

If the alias relation r' obtained after the successful execution of p (marked by $\boxed{\text{l}}$ at the top of the continuation) is not a lasso of the aliasing r before p (previously stored in $\langle-\rangle_{\mathrm{bkt\text{-}l}}$) then p is constrained to a new execution by becoming the top of the k-cell, and r' is memorized for back-tracking:

$$\langle\ r'\ \rangle_{\mathrm{al}}\ \Big\langle\ \frac{\boxed{\text{l}}\ \textbf{loop } p \textbf{ end}}{p\ \boxed{\text{l}}\ \textbf{loop } p \textbf{ end}}\ \ldots\Big\rangle_{\mathrm{k}}\ \Big\langle\ \frac{\langle\ r\ \rangle_{\mathrm{al\text{-}o}}\langle\ p\ \rangle_{\mathrm{l}}}{\langle\ r'\ \rangle_{\mathrm{al\text{-}o}}\langle\ p\ \rangle_{\mathrm{l}}}\ \ldots\Big\rangle_{\mathrm{bkt\text{-}l}}\ \text{ if not lasso}(r,r')\ (\clubsuit) \tag{30}$$

Last, if a lasso is reached after the execution of p, then the current aliasing is soundly replaced by a "regular" over-approximation $\text{reg}(r, r')$, the corresponding back-tracking information is removed from $\langle - \rangle_{\text{bkt-l}}$ and the **loop** instruction is eliminated from the k-cell:

$$\langle \; \frac{r'}{\text{reg}(r,r')} \; \rangle_{\text{al}} \langle \; \frac{\boxed{1} \; \textbf{loop} \; p \; \textbf{end}}{.} \ldots \rangle_{\text{k}} \langle \; \frac{\langle \; r \; \rangle_{\text{al-o}} \langle \; p \; \rangle_{\text{l}}}{.} \ldots \rangle_{\text{bkt-l}} \; \text{if lasso}(r,r') \; (\clubsuit) \quad (31)$$

For handling function calls such as **call** $f(l)$ we use a meta-construction $| \; f \; | \; \boxed{f}$. Here $| \; f \; |$ stands for the body of f and $\boxed{f}$ marks the end of the corresponding execution. Moreover, a stack $\langle - \rangle_{\text{bkt-cf}}$ is utilized in order to store the alias information before each (possibly recursive) call of f, with the purpose of identifying the lassos generated by the (possibly repeated) execution of f. In order to guarantee a sound implementation of (mutually) recursive calls, both $\boxed{f}$ and $\langle - \rangle_{\text{bkt-cf}}$ are parameterized by f – the name of the function. An example illustrating this reasoning mechanism is provided in Appendix D.

The first $\mathbb{K}$ rule for handling function calls matches the associated axiom in (8): the alias information is set to $r[f^{\bullet}{:}l]$, whereas the next instructions to be executed are given by $| \; f \; |$. Note that the original aliasing is retained in the (initially empty) back-tracking cell via $\langle r \rangle_{\text{al-o}}$.

$$\langle \; \frac{r}{r[f^{\bullet}{:}l]} \; \rangle_{\text{al}} \langle \; \frac{\textbf{call} \; f(l)}{| \; f \; | \; \boxed{f}} \ldots \rangle_{\text{k}} \langle \; \frac{.}{\langle \; r \; \rangle_{\text{al-o}}} \; \rangle_{\text{bkt-cf}} \quad (32)$$

Remark 4. Observe that the back-tracking cell does not need to be parameterized by the actual argument list l of f. Each such argument is anyways replaced in the current alias relation r by its counterpart in the formal argument list of f. In short: r becomes $r[f^{\bullet}{:}l]$.

A successful execution of **call** $f(l)$ is distinguished by the occurrence of $\boxed{f}$ at the top of the continuation stack. If this is the case, then the corresponding back-tracking alias information is removed from $\langle - \rangle_{\text{bkt-cf}}$ and the next program instruction (if any) is enabled at the top of the k-cell:

$$\langle \; r' \; \rangle_{\text{al}} \langle \; \frac{\boxed{f}}{.} \ldots \rangle_{\text{k}} \langle \; \frac{\langle \; r \; \rangle_{\text{al-o}}}{.} \ldots \rangle_{\text{bkt-cf}} \quad (33)$$

Recursive calls are treated by means of two $\mathbb{K}$-rules. Note that a recursive context is identified whenever the current program instruction is of shape **call** $f(l)$ and the associated back-tracking structure is not empty, *i.e.*, rule (32) was previously applied. Then, if the recursive call of f when starting with r produces a lasso r', the execution of $f(l)$ is stopped by soundly over-approximating the alias information with $\text{reg}(r, r')$, according to Lemma 2, and by removing **call** $f(l)$ from the k-cell:

$$\langle \; \frac{r'}{\text{reg}(r,r')} \; \rangle_{\text{al}} \langle \; \frac{\textbf{call} \; f(l)}{.} \ldots \rangle_{\text{k}} \langle \; \langle \; r \; \rangle_{\text{al-o}} \; \ldots \rangle_{\text{bkt-cf}} \; \text{if lasso}(r,r') \; (\clubsuit) \quad (34)$$

If a lasso is not reached, then the body of f is executed once more, and the current aliasing is pushed to the back-tracking cell:

$$\langle\ r'\ \rangle_{\text{al}}\ \frac{\langle\ \textbf{call } f(l)\ \ldots\rangle_{\text{k}}}{\mid f\mid\ \boxed{\text{f}}}\ \langle\ \frac{.}{\langle\ r'\ \rangle_{\text{al-o}}}\frac{\langle\ r\ \rangle_{\text{al-o}}\ \ldots\rangle_{\text{bkt-cf}}}{}\ \text{if not } \text{lasso}(r,r')\ \ (\clubsuit) \qquad (35)$$

Qualified calls $x.\textbf{call } f(l)$ are handled by two $\mathbb{K}$-rules as follows. First, based on the definition in (8), the "negative variable" x' transposing the context of the call to to the context of the caller is distributed to the elements of the initial alias relation r, and to l – the argument list of f. Moreover, a meta-construction $\boxed{\text{qf}}$ is utilized in order to mark the end of the qualified call in the continuation cell, similarly to the rule (32). The caller is stored in a back-tracking stack $\langle\ .\ \rangle_{\text{bkt-qf}}$ also parameterized by f – the name of the function. The current instruction in the k-cell becomes $\textbf{call } f(x'.l)$, as expected:

$$\langle\ \frac{r}{x'.r}\ \rangle_{\text{al}}\ \langle\ \frac{x.\textbf{call } f(l)}{\textbf{call } f(x'.l)\ \boxed{\text{qf}}}\ \ldots\rangle_{\text{k}}\ \langle\ \frac{.}{\langle\ x\ \rangle_{\text{f}}}\ \rangle_{\text{bkt-qf}} \qquad (36)$$

Second, when the successful termination of the qualified call is signaled by $\boxed{\text{qf}}$ at the top of the k-cell, the corresponding stored caller is distributed to the current alias relation and removed from the back-tracking cell. The next instruction in the continuation cell is released by eliminating the top $\boxed{\text{qf}}$:

$$\langle\ \frac{r}{x.r}\ \rangle_{\text{al}}\ \langle\ \frac{\boxed{\text{qf}}}{.}\ \ldots\rangle_{\text{k}}\ \langle\ \frac{\langle\ x\ \rangle_{\text{f}}}{.}\ \ldots\rangle_{\text{bkt-qf}} \qquad (37)$$

D The $\mathbb{K}$-machinery by example

For an example, in this section we show how the $\mathbb{K}$-machinery developed in Section 3 can be used in order to extract the alias information for the case of two mutually recursive functions defined as:

$$f(x)\ \{\ x := x.a\,;\ \ \textbf{call } g(x)\ \} \qquad\qquad g(x)\ \{\ x := x.b\,;\ \ \textbf{call } f(x)\ \}$$

We assume that x is an object of a class with two fields a and b, respectively. We consider a sequential setting.

At first glance it is easy to see that the execution of $\textbf{call } f(x)$, when starting with an empty alias relation r, produces the alias expressions:

$$[x,\ x.(a.b)^*]\quad [x.a,\ x.(a.b)^*.a]\quad [x.b,\ x.(a.b)^*.b] \qquad (38)$$

The associated reasoning in $\mathbb{K}$ is depicted in the figure below. The whole procedure starts with an empty alias relation $r = \emptyset$, and $\textbf{call } f(x)$ in the continuation stack. Then, the corresponding $\mathbb{K}$ rules (for handling assignments and function calls) are applied in the natural way.

A lasso is reached after two calls of $f(x)$ that, consequently, determine two calls of $g(x)$ – identified by $\boxed{g}$ $\boxed{f}$ $\boxed{g}$ $\boxed{f}$ in the k-cell. This triggers the application of rule (34) enabling the "regular" over-approximation as in Lemma 2.

Our example also illustrates the importance of isolating the back-traced alias information in cells of shape $\langle\, .\, \rangle_{\text{bkt-cf}}$ parameterized by the (possibly recursive) function f. More explicitly, rule (34) is soundly applied by identifying the aforementioned lasso based on: the current alias relation r_4, the recursive call $f(l)$ at the top of the continuation, and the back-traced aliasing $\langle\, \langle\, r_2\, \rangle_{\text{al-o}}\, \ldots \rangle_{\text{bkt-cf}}$ associated to the previous executions of $f(l)$.

As introduced in (12), an alias relation r' is a lasso of a relation r whenever there is a one-to-one correspondence between their elements as follows:

$$[x_1y_1z_1, x_2y_2z_2] \in r \quad \text{iff} \quad [x_1y_1y_1z_1, x_2y_2y_2z_2] \in r'.$$

The current alias relation

$$r_4 = \{[x, x.a.b.a.b],\ [x.a, x.a.b.a.b.a],\ [x.b, x.a.b.a.b.b]\},$$

before applying rule (34), is a lasso of

$$r_2 = \{[x, x.a.b],\ [x.a, x.a.b.a],\ [x.b, x.a.b.b]\}.$$

The aforementioned one-to-one correspondence is summarized in the following table:

$[x_1y_1z_1, x_2y_2z_2] \in r_2$ iff $[x_1y_1y_1z_1, x_2y_2y_2z_2] \in r_4$	x_1	y_1	z_1	x_2	y_2	z_2
$[x, x.a.b] \in r_2$ iff $[x, x.a.b.a.b] \in r_4$	x	ε	ε	x	$a.b$	ε
$[x.a, x.a.b.a] \in r_2$ iff $[x.a, x.a.b.a.b.a] \in r_4$	x	ε	a	x	$a.b$	a
$[x.b, x.a.b.b] \in r_2$ iff $[x.b, x.a.b.a.b.b] \in r_4$	x	ε	b	x	$a.b$	b

Here ε stands for the *empty alias expression*.

Moreover, according to rule (34), the lasso shaped by r_2 and r_4 also causes the (otherwise infinite) recursive calls to stop, as **call** $f(l)$ is eliminated from the top of the k-cell. Hence, the rewriting process finishes with a sound over-approximation $\text{reg}(r_2, r_4)$ replacing the current alias relation (cf. Lemma 2), defined precisely as in (38).

$\langle\, r\,\rangle_{\mathrm{al}}\ \langle\ \mathbf{call}\ f(x)\ \rangle_{\mathrm{k}}$
$\langle\,\cdot\,\rangle_{\mathrm{bkt\text{-}cf}}\ \langle\,\cdot\,\rangle_{\mathrm{bkt\text{-}cg}}$

$$\Downarrow (32)$$

$\langle\, r\,\rangle_{\mathrm{al}}\ \langle\ x := x.a;\mathbf{call}\ g(x)\ \boxed{\mathrm{f}}\ \rangle_{\mathrm{k}}$
$\langle\ \langle\, r\,\rangle_{\mathrm{al\text{-}o}}\,\rangle_{\mathrm{bkt\text{-}cf}}\ \langle\,\cdot\,\rangle_{\mathrm{bkt\text{-}cg}}$

$$\Downarrow (25)$$

$\langle\, r_1\,\rangle_{\mathrm{al}}\ \langle\ \mathbf{call}\ g(x)\ \boxed{\mathrm{f}}\ \rangle_{\mathrm{k}}$
$\langle\ \langle\, r\,\rangle_{\mathrm{al\text{-}o}}\,\rangle_{\mathrm{bkt\text{-}cf}}\ \langle\,\cdot\,\rangle_{\mathrm{bkt\text{-}cg}}$
where $r_1 = \{[x, x.a], [x.a, x.a.a], [x.b, x.a.b]\}$

$$\Downarrow (35)$$

$\langle\, r_1\,\rangle_{\mathrm{al}}\ \langle\ x := x.b;\mathbf{call}\ f(x)\ \boxed{\mathrm{g}}\ \boxed{\mathrm{f}}\ \rangle_{\mathrm{k}}$
$\langle\ \langle\, r\,\rangle_{\mathrm{al\text{-}o}}\,\rangle_{\mathrm{bkt\text{-}cf}}\ \langle\ \langle\, r_1\,\rangle_{\mathrm{al\text{-}o}}\,\rangle_{\mathrm{bkt\text{-}cg}}$

$$\Downarrow (25)$$

$\langle\, r_2\,\rangle_{\mathrm{al}}\ \langle\ \mathbf{call}\ f(x)\ \boxed{\mathrm{g}}\ \boxed{\mathrm{f}}\ \rangle_{\mathrm{k}}$
$\langle\ \langle\, r\,\rangle_{\mathrm{al\text{-}o}}\,\rangle_{\mathrm{bkt\text{-}cf}}\ \langle\ \langle\, r_1\,\rangle_{\mathrm{al\text{-}o}}\,\rangle_{\mathrm{bkt\text{-}cg}}$
where $r_2 = \{[x, x.a.b], [x.a, x.a.b.a], [x.b, x.a.b.b]\}$

$$\Downarrow (35)$$

$\langle\, r_2\,\rangle_{\mathrm{al}}\ \langle\ x := x.a;\mathbf{call}\ g(x)\ \boxed{\mathrm{f}}\ \boxed{\mathrm{g}}\ \boxed{\mathrm{f}}\ \rangle_{\mathrm{k}}$
$\langle\ \langle\, r_2\,\rangle_{\mathrm{al\text{-}o}}\ \langle\, r\,\rangle_{\mathrm{al\text{-}o}}\,\rangle_{\mathrm{bkt\text{-}cf}}\ \langle\ \langle\, r_1\,\rangle_{\mathrm{al\text{-}o}}\,\rangle_{\mathrm{bkt\text{-}cg}}$

$$\Downarrow (25)$$

$\langle\, r_3\,\rangle_{\mathrm{al}}\ \langle\ \mathbf{call}\ g(x)\ \boxed{\mathrm{f}}\ \boxed{\mathrm{g}}\ \boxed{\mathrm{f}}\ \rangle_{\mathrm{k}}$
$\langle\ \langle\, r_2\,\rangle_{\mathrm{al\text{-}o}}\ \langle\, r\,\rangle_{\mathrm{al\text{-}o}}\,\rangle_{\mathrm{bkt\text{-}cf}}\ \langle\ \langle\, r_1\,\rangle_{\mathrm{al\text{-}o}}\,\rangle_{\mathrm{bkt\text{-}cg}}$
where $r_3 = \{[x, x.a.b.a], [x.a, x.a.b.a.a], [x.b, x.a.b.a.b]\}$

$$\Downarrow (35)$$

$\langle\, r_3\,\rangle_{\mathrm{al}}\ \langle\ x := x.b;\mathbf{call}\ f(x)\ \boxed{\mathrm{g}}\ \boxed{\mathrm{f}}\ \boxed{\mathrm{g}}\ \boxed{\mathrm{f}}\ \rangle_{\mathrm{k}}$
$\langle\ \langle\, r_2\,\rangle_{\mathrm{al\text{-}o}}\ \langle\, r\,\rangle_{\mathrm{al\text{-}o}}\,\rangle_{\mathrm{bkt\text{-}cf}}\ \langle\ \langle\, r_3\,\rangle_{\mathrm{al\text{-}o}}\ \langle\, r_1\,\rangle_{\mathrm{al\text{-}o}}\,\rangle_{\mathrm{bkt\text{-}cg}}$

$$\Downarrow (25)$$

$\langle\, r_4\,\rangle_{\mathrm{al}}\ \langle\ \mathbf{call}\ f(x)\ \boxed{\mathrm{g}}\ \boxed{\mathrm{f}}\ \boxed{\mathrm{g}}\ \boxed{\mathrm{f}}\ \rangle_{\mathrm{k}}$
$\langle\ \langle\, r_2\,\rangle_{\mathrm{al\text{-}o}}\ \langle\, r\,\rangle_{\mathrm{al\text{-}o}}\,\rangle_{\mathrm{bkt\text{-}cf}}\ \langle\ \langle\, r_3\,\rangle_{\mathrm{al\text{-}o}}\ \langle\, r_1\,\rangle_{\mathrm{al\text{-}o}}\,\rangle_{\mathrm{bkt\text{-}cg}}$
where $r_4 = \{[x, x.a.b.a.b], [x.a, x.a.b.a.b.a], [x.b, x.a.b.a.b.b]\}$

$$\Downarrow (34)$$

$\langle\ reg(r_2, r_4)\,\rangle_{\mathrm{al}}\ \langle\ \boxed{\mathrm{g}}\ \boxed{\mathrm{f}}\ \boxed{\mathrm{g}}\ \boxed{\mathrm{f}}\ \rangle_{\mathrm{k}}$
$\langle\ \langle\, r_2\,\rangle_{\mathrm{al\text{-}o}}\ \langle\, r\,\rangle_{\mathrm{al\text{-}o}}\,\rangle_{\mathrm{bkt\text{-}cf}}\ \langle\ \langle\, r_3\,\rangle_{\mathrm{al\text{-}o}}\ \langle\, r_1\,\rangle_{\mathrm{al\text{-}o}}\,\rangle_{\mathrm{bkt\text{-}cg}}$

$$\Downarrow (*)(33)$$

$\langle\ \{[x, x.(a.b)^*], [x.a, x.(a.b)^*.a], [x.b, x.(a.b)^*.b]\}\,\rangle_{\mathrm{al}}\langle\,\cdot\,\rangle_{\mathrm{k}}\langle\,\cdot\,\rangle_{\mathrm{bkt\text{-}cf}}\langle\,\cdot\,\rangle_{\mathrm{bkt\text{-}cg}}$

Fig. 1. Aliasing and mutual recursion in $\mathbb{K}$.

E   Example of aliasing in Maude

The intermediate class-based representation (in `aliasing-linked_list.maude`) corresponding to the example

$$x := y;$$
$$\textbf{loop } x := x.next \textbf{ end}$$

is given as:

```
(class 'LINKED_LIST_TEST
    create  { 'make }
    (
        procedure { 'ANY } 'make ( nil )
            require True
            local
                (  'x : [?, . , 'NODE] ;  'y : [?, . , 'NODE] ;  )
            do
                (
                assign ('x, 'y);
                until False loop ( assign ('x, 'x . 'next(nil)) ; ) end ;
                )
            ensure True
            rescue nil
        end ;
    )

    invariant  True
end) ;

class 'NODE
    create  {'make}
    (
    attribute { 'ANY } 'next : [?, . , 'NODE] ;
    )

    invariant True
end ;
```

As can be seen from the code above, the syntax enables expressing Eiffel-like properties of classes by using assertions s.a. preconditions (introduced by the keyword **require**), postconditions (through the keyword **ensure**) and class invariants.

The "entry point" of the program corresponds to the function `'make` in the (main) class `'LINKED_LIST_TEST` and is set via:

```
settings('LINKED_LIST_TEST, 'make, false, aliasing-on) .
```

Observe that the flag for performing the alias analysis is switched to "on". In `'LINKED_LIST_TEST`, two local variables x and y of type `NODE` are declared as

running on the current processor $(.)$, *i.e.*, they are not *separate.* The instruction `assign('x, 'y)`, for instance, corresponds to the assignment $x := y$. The class defining a `NODE` structure (in a linked list) simply consists of a (non-separate) field `'next` of type `NODE`.

We run the example by executing the command:

```
> maude SCOOP.maude ..\examples\aliasing-linked_list.maude
```

The relevant parts of the corresponding Maude output are as follows:

```
                 \||||||||||||||||||/
                 --- Welcome to Maude ---
                 /||||||||||||||||||\
             Maude 2.6 built: Mar 31 2011 23:36:02
             Copyright 1997-2010 SRI International
[...]
=========================================
rewrite [100] in SYSTEM :
[...]
{0}proc(1) ::
until False loop
  assign('x, 'x . 'next(nil)) ;
end ;
[...],
100,
aliasing-on
({['x ; 'y . 'next . 'next . 'next . 'next .
    'next . 'next . 'next . 'next . 'next . 'next . 'next . 'next .
    'next . 'next . 'next . 'next . 'next . 'next . 'next . 'next .
    'next . 'next . 'next . 'next . 'next . 'next . 'next . 'next .
    'next . 'next . 'next . 'next . 'next . 'next . 'next . 'next .
    'next . 'next . 'next . 'next . 'next . 'next]} U
 {['x . 'next ; 'y . 'next . 'next . 'next . 'next . 'next . 'next .
    'next . 'next . 'next . 'next . 'next . 'next . 'next . 'next .
    'next . 'next .  'next . 'next . 'next . 'next . 'next . 'next .
    'next . 'next . 'next . 'next . 'next . 'next . 'next . 'next .
    'next . 'next . 'next . 'next . 'next . 'next . 'next . 'next .
    'next . 'next . 'next . 'next . 'next]})
[...]
state
  [...]
  heap [...]
  store [...]
end
```

In short, after 100 rewriting steps, the current processor `{0}proc(1)` has the execution corresponding to **loop** $x := x.next$ **end** on top of its instruction stack, and the aliasing information contains (the dot-complete closure of) the relation $\{[x, y.next^{42}]\}$. Moreover, the output displays the contents of the current system state, by providing information on the *heap* and *store*, as formalized in [21].