On the Verification of SCOOP Programs

Georgiana Caltais, Bertrand Meyer

Department of Computer Science, ETH Zürich, Switzerland
Software Engineering Lab, Innopolis University, Russia

Abstract

In this paper we focus on the development of a unifying framework for the formal modelling of an object oriented-programming language, its underlying concurrency model and their associated analysis tools. More precisely, we target SCOOP – an elegant concurrency model, recently formalized based on Rewriting Logic (RL) and Maude. SCOOP is implemented in Eiffel and its applicability is demonstrated also from a practical perspective, in the area of robotics programming. Our contribution consists in devising and integrating an alias analyzer and a Coffman deadlock detector under the roof of the same RL-based semantic framework of SCOOP. This enables using the Maude rewriting engine and its LTL model-checker “for free”, in order to perform the analyses of interest. We discuss the limitations of our approach for model-checking deadlocks and provide possible workarounds for the state explosion problem. On the aliasing side, 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 easily implementable on top of the SCOOP formalization. An important property of our extension is that, in non-concurrent settings, the corresponding alias expressions can be over-approximated in terms of a notion of regular expressions. This further enables us to derive 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.

Keywords: concurrency, SCOOP, operational semantics, alias analysis, deadlock detection, Maude, rewriting logic

1. Introduction

In light of the widespread deployment and complexity of concurrent systems, the development of corresponding frameworks for rigorous design and analysis has been a great challenge. In this paper we focus on the development of a unifying framework for the formal modelling of an object oriented-programming language, its underlying concurrency model and their associated analysis tools.

We are targeting SCOOP [1], 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, and 2) the burden of orchestrating concurrent executions is handled within the model, therefore reducing the risk of correctness issues. The reference implementation is Eiffel [2], but implementations have also been built on top of Java-like languages [3]. The success of SCOOP is demonstrated not only from a research perspective, but also from a practical perspective, with applications appearing, for instance, in the area of robotics programming [4].

The basis of a framework for the design and analysis of the SCOOP model has already been set. In this respect, we refer to the recent formalization of SCOOP in [1] based on Rewriting Logic (RL) [5], which is executable and straightforwardly implementable in the programming language Maude [6]. In [1] these
capabilities have been successfully exploited in order to reason on the original SCOOP model and to identify a number of design flaws.

Moreover, an executable semantics can be exploited in order to formalize and “run” analysis tools for SCOOP programs as well. This facilitates the extension of the aforementioned SCOOP formalization to the level of a unifying executable semantic framework for the design and analysis of both the model and its concurrent applications. In this paper we focus on the development of a RL-based toolbox for the analysis of SCOOP programs on top of the formalization in [1]. We are interested in constructing an alias analyzer and a deadlock detector.

Alias analysis has been an interesting research direction for the 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 [7, 8] that handle isolated functions only, and their inter-procedural counterparts [8, 9, 10] that consider the interactions between function calls; (ii) type-based techniques [11]; (iii) flow-based techniques [12, 13] that establish aliases depending on the control-flow information of a procedure; (iv) context-(in)sensitive approaches [14, 15] that depend on whether the calling context of a function is taken into account or not; (v) field-(in)sensitive approaches [16, 17] that depend on whether the individual fields of objects in a program are traced or not.

There is a huge literature on heap analysis for aliasing [18], but hardly any paper that presents a calculus allowing the derivation of alias relations as the result of applying various instructions of a programming language. Hence, 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 [19]. The 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 [19] abstracts the aliasing information in terms of explicit access paths [20] referred to as alias expressions straightforwardly computed in an equational fashion, based on the language constructs. As we shall see later on in this paper, the language-based expressions can be exploited in order to reason on “may aliasing” in a finite number of steps in non-concurrent settings and, moreover, can be easily incorporated in the semantic rules defining SCOOP in [1].

The work most similar to our contribution is the one in [21], where aliasing information is identified by exploiting regular behaviour of (non-concurrent) programs, in a RL setting. The main difference with the results in [21] consists in how the abstract memory addresses corresponding to pointer variables are represented. In [21] these range over a finite set of natural numbers. In this paper we consider alias expressions build according to the calculus in [19], based on program constructs.

In [22] the authors focus on alias analysis and type inference for PHP and sketch a hypothetical solution of their approach in Maude. The alias analysis in [22] is inter-procedural as well, and can handle function pointers, recursive calls and references. One of the major differences with our work is that in [22], in order to derive correct analysis results, the authors run aliasing and type inference analysis in “tandem” until a fixpoint is reached. We guarantee the termination our analysis by exploiting a sound over-approximation of aliasing via the so-called regular alias expressions.

Deadlock is one of the most serious problems in concurrent systems. It occurs when two or more executing threads are each waiting for the other to finish. Along time, the complexity of the problem determined various approaches to combat deadlocks [23]. Examples include: (i) deadlock prevention [24] which ensures that at least one of the deadlock conditions cannot hold, (ii) deadlock avoidance [25] that provides a priori information so that the system can predict and avoid deadlock situations, (iii) deadlock detection [26, 27] that detects and recovers from a deadlock state.

Our focus is on deadlock detection for SCOOP programs. We base our work on the fact that this type of analysis is in strict connection with the underlying model of interest. Consequently, as described in the corresponding subsequent sections, our approach consists in formalizing deadlocks in the context of the SCOOP concurrency model and enriching its semantics in [1] with the equivalent operational-based definition of deadlocks. This enables using the Maude rewriting capabilities “for free” in order to test SCOOP programs for deadlock. Nevertheless, the more ambitious goal of using the Maude LTL model-checker for
deadlock detection is not straightforward. As discussed in more detail later on in this paper, verification of deadlocks was possible after reducing the SCOOP semantics in [1] and abstracting it based on aliasing information, and modifying a series of implementation aspects (such as indexed-based parameterizations) that determined state explosion issues.

The literature on using static analysis [8] and abstracting techniques for (related) concurrency models is considerable. We refer, for instance, to the recent work in [46] that introduces a framework for detecting deadlocks by identifying circular dependencies in the (finite state) model of so-called contracts that abstract methods in an OO-language. Nevertheless, the integration of a deadlock analyzer in SCOOP on top of Maude is an orthogonal approach that aims at constructing a RL-based toolbox for SCOOP programs laying over the same semantic framework.

In [47] SCOOP programs are verified for deadlocks and other behavioral properties using GROOVE [48]. The work in [47] proposes a redefinition of the most common features of the SCOOP semantics based on graph transformation systems (GPSs). This is a bottom-up approach, as it aims at redefining the SCOOP semantics from scratch via GPSs, orthogonal to our rather top-down strategy of narrowing the original semantics proposed in [1].

There are several RL-based approaches to deadlock detection in the literature. In [28, 29], for instance, the authors use the Maude LTL model checker to show deadlock-freedom in a dining philosophers implementation in C. In [30], the Maude LTL model checker is used to verify safety and liveness properties of Orc [31] programs such as a dining philosophers implementation and an online auction system. Nevertheless, none of the aforementioned results are used in the context of a unifying framework for specifying and clarifying a concurrency model [1], and analyzing its corresponding applications.

The results in this paper centre around SCOOP as tackling aliasing and deadlock related aspects is particularly challenging in the context of a concurrency model as complex as SCOOP. However, exploiting a unifying RL-based framework for both designing the concurrency model and analyzing its corresponding applications raises certain issues that were not obvious at the stage of just formalizing SCOOP in [1]. Some of the identified design and implementation decision affecting the verification of SCOOP applications in a negative way can serve as guidelines in the context of other RL-based models as well.

This paper is an extended version of [32] where we proposed:

1. a translation of the (finite) alias calculus in [19] to the setting of unbounded program executions such as infinite loops and recursive calls, together with a sound over-approximation technique based on (finitely representable) “regular alias expressions” capturing unbounded executions in non-concurrent settings;
2. a RL-based specification of the extended calculus suitable for integration within the SCOOP formalization in [1] (for this purpose we use a notation inspired by the K-framework enabling compact and modular definitions);
3. an algorithm for “may aliasing” (exploiting the finiteness property in 1.) that always terminates in non-concurrent settings.

The current work adds to 1.–3. above:

4. the full RL-based specification in 2. and the complete formal proofs showing the soundness of the over-approximating technique based on “regular alias expressions”;
5. examples of exploiting the algorithm in 3. and its implementation on top of the SCOOP formalization in Maude [1];
6. the formalization and integration of a deadlock detection mechanism on top of the SCOOP operational semantics [1], together with discussions on the limitations of our approach and associated workarounds.

Paper structure. The paper is organized as follows. In Section 2 we provide a brief overview of SCOOP. In Section 3 we introduce the extension of the alias calculus in [19] to unbounded executions. In Section 4 we provide the full RL-based executable specification of the calculus. The implementation in SCOOP and further applications are discussed in Section 5. Section 6 is dedicated to deadlocking in SCOOP. In Section 7 we draw the conclusions and provide pointers to future developments.
2. Brief introduction to SCOOP

As already stated, the purpose of the current work is the development of a toolbox for the analysis of SCOOP programs by exploiting the semantics proposed in [1]. SCOOP is particularly attractive due to its simplicity and elegance, as it allows the switch from sequential to concurrent programming in a rather straightforward fashion, by means of just one keyword, namely, separate. Transparent to the user, the key notion in SCOOP is the processor, or handler (that can be a CPU, or it can also be implemented in software, as a process or thread). Handlers are in charge of executing the routines of “separate” objects, in a concurrent fashion.

For an example, assume a processor \( p \) that performs a call \( o.f(a_1,a_2,\ldots) \) on an object \( o \). If \( o \) is declared as “separate”, \( p \) sends a request for executing \( f(a_1,a_2,\ldots) \) to \( q \) – the handler of \( o \) (note that \( p \) and \( q \) can coincide). Meanwhile, \( p \) can continue. Moreover, assume that \( a_1,a_2,\ldots \) are of “separate” types. According to the SCOOP semantics, the application of the call \( f(\ldots) \) will wait until it has been able to lock all the separate objects associated to \( a_1,a_2,\ldots \). This mechanism guarantees exclusive access to these objects. Given a processor \( p \), by \( W(p) \) we denote the set of processors \( p \) waits to release the resources \( p \) needs for its asynchronous execution. Orthogonally, by \( H(p) \) we represent the set of resources (more precisely, resource handlers that) \( p \) already acquired.

The semantics of SCOOP in [1] is defined over tuples of shape

\[
\langle p_1::St_1 | \cdots | p_n::St_n, \sigma \rangle
\]

(1)

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

Roughly speaking, one could classify the operational rules formalizing SCOOP in [1] in: a) language rules that provide the semantics of language constructs such as “\( \text{if} \ldots \text{then} \ldots \text{else} \ldots \text{end} \)” or “\( \text{until} \ldots \text{loop} \ldots \text{end} \)”, and b) control rules implementing mechanisms such as locking or scheduling.

For an example in category a) above, consider the rules specifying “if” instructions:

\[
\frac{a \text{ is fresh}}{(p::\text{if } e \text{ then } St_1 \text{ else } St_2 \text{ end}; St, \sigma) \rightarrow (p::\text{eval}(a,e); \text{wait}(a); \text{provided } a.data \text{ then } St_1 \text{ else } St_2; St, \sigma)}
\]

(2)

\[
\frac{}{(p::\text{eval}(a,e); \text{wait}(a); \text{provided } a.data \text{ then } St_1 \text{ else } St_2; St, \sigma) \rightarrow (p::St_1; St, \sigma)}
\]

(3)

\[
\frac{}{(p::\text{provided } true \text{ then } St_1 \text{ else } St_2; St, \sigma) \rightarrow (p::St_1; St, \sigma)}
\]

(4)

Intuitively, “\( \text{eval}(a,e) \)” evaluates \( e \) and puts the result on a fresh channel \( a \) and “\( \text{wait}(a) \)” enables processor \( p \) to use the evaluation result stored in \( a.data \). It is straightforward to see that, according to (3), in case the condition \( e \) is evaluated to \( \text{true} \) then the “if branch” \( St_1 \) is placed on top of the call stack of \( p \). Otherwise, based on (4), if \( e \) is evaluated to \( \text{false} \), the “else branch” is executed.

As we shall see in Section 5, an operational view on the alias calculus in [19] exploiting the instructions of a programming language will enable a straightforward implementation on top of the “language rules” of SCOOP.

For the case b) above we refer to the locking rule:

\[
\frac{}{(p::\text{lock}([q_1,\ldots,q_m]); St, \sigma) \rightarrow (p::St, \sigma,\text{lock}_\text{rqqs}(p, [q_1,\ldots,q_m])); \forall q_i \in \{q_1,\ldots,q_m\}: \sigma,\text{rq\_locked}(q_i) = \text{false}}
\]

(5)

stating that a processor \( p \) can lock a set of handlers \( \{q_1,\ldots,q_m\} \) by calling \( \text{lock\_rqqs} \) on the state \( \sigma \) whenever none of the handlers \( q_i \) has previously been acquired by other processors, i.e., \( \sigma,\text{rq\_locked}(q_i) = \text{false} \).

As it will become clear in Section 6, “control rules” pave the way to an immediate implementation of a corresponding “deadlock rule” on top of the Maude formalization of SCOOP in [1].
3. The alias calculus

The calculus for may aliasing introduced in [19] abstracts the aliasing information in terms of explicit access paths referred to as “alias expressions”. Consider, for an example, the case of a linked list. We write \( x_i (i \geq 0) \) to represent node \( i \) in the list, and use a setter to assign the next node of the list:

\[
\text{create } x_0 \\
\text{loop} \quad i := i + 1 \\
\text{create } x_i \\
x_i.set\_next(x_{i-1}) \\
\text{end}
\]  

(6)

The result of the execution of the code above can be intuitively depicted as the infinite sequence:

\[
x_0 \xleftarrow{\text{next}} x_1 \xleftarrow{\text{next}} \ldots x_{k-1} \xleftarrow{\text{next}} x_k \xleftarrow{\text{next}} x_{k+1} \ldots
\]

Hence, \( x_0 \) becomes aliased to \( x_1.next, x_2.next.next, x_3.next.next.next \), so on and so on. In short, the set of associated alias expressions can be equivalently written as:

\[
\{ [x_i, x_i+k.next^k] | i \geq 0 \land k \geq 1 \}.
\]

(7)

The sources of imprecision introduced by the calculus in [19] are limited to ignoring tests in conditionals, and to “cutting at length \( L \)” for the case of possibly infinite alias relation corresponding to unbounded executions as in (6). The cutting technique considers sequences longer than a given length \( L \) as aliased to all expressions.

In this section we define an extension of the calculus in [19], to unbounded program executions. Moreover, based on the idea behind the pumping lemma for regular languages [33], 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 4. The machinery for collecting aliases is defined using a notation inspired by the K framework [34], as its operational flavor enables a straightforward integration within the SCOOP formalization in [1].

We proceed by recalling the notion of alias relation and a series of associated notations and basic operations, as introduced in [19].

**Definition 1** (Alias expressions [19]). We call an alias 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 \( e.Current = \text{Current}.e = e \).

**Definition 2** (Alias relations [19]). Let \( E \) represent the set of all expressions of a program. An alias relation is a symmetric and irreflexive binary relation \( r \subseteq E \times E \). Let \([e_1,e_2] \in r\); we call \( [e_1,e_2] \) an alias pair in \( r \).

Given an alias relation \( r \) and an expression \( e \), we define

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

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 \( x.e \):

\[
r - x = \{[e_1,e_2] \mid \forall e : E . [e_1,e_2] \in r \land e_1 \not= x.e \land e_2 \not= x.e\}.
\]

By \( a \in \text{dom}(t) \) we refer to an attribute in the class corresponding to the object referred by the alias 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 \( t = x.next.next \).
Intuitively, we say that an alias relation is dot-complete, whenever its elements are soundly extended via fields in their corresponding domains. The operation making a relation dot-complete is defined as follows:

\[
dotcomplete(r) = r \cup \{[t.a, s.a] \mid [t, s] \in r \land a \in \text{dom}(t)\} \cup \{[s.a, v] \mid [t, s] \in r \land [t.a, v] \in r\}.
\]

Let \(x\) be an alias expression and \(u\) be a set of expressions in \(E\). We define the relation \(r\) augmented with pairs \([x, y]\), for \(y \in u\), and made dot complete as

\[
r[x = u] = \dotcomplete(r') \text{ where } r' = r \cup \{[x, y] \mid y \in u\}.
\]

3.1. Extension to unbounded executions

We further introduce an extension of the alias calculus in [19] 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 [19], the calculus ignores tests in conditionals and loops.

**Definition 3** (Program instructions [19]). Let \(x, x'\) be variable names and \(s\) a valid sequence \(x'.y_1.y_2\ldots.y_n\) with \(y_1 \in \text{dom}(x')\), \(y_2 \in \text{dom}(x'.y_1)\), \ldots, \(y_n \in \text{dom}(x'.y_1\ldots.y_{n-1})\). Let \(f\) be a routine name and \(l\) the actual parameters of a call of \(f\). The program instructions are defined as follows:

\[
p ::= p; p \mid \text{then } p \text{ else } p \text{ end} \mid \text{create } x \mid \text{forget } x \mid x := s \mid \text{loop } p \text{ end} \mid \text{call } f(l) \mid x.\text{call } f(l).
\]

We write \(r \triangleright p\) to represent the alias information obtained by executing \(p\) when starting with the initial alias relation \(r\).

We further give the axioms of the extended alias calculus. The axiom for sequential composition is defined in the obvious way:

\[
r \triangleright (p; q) = (r \triangleright p) \triangleright q.
\]

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:

\[
r \triangleright (\text{then } p \text{ else } q \text{ end}) = r \triangleright p \cup r \triangleright q.
\]

As previously mentioned, we define \(r \triangleright \text{loop } p \text{ end}\) according to its informal semantics: “execute \(p\) repeatedly any number of times, including zero”. The corresponding rule is:

\[
r \triangleright (\text{loop } p \text{ end}) = \bigcup_{n \in \mathbb{N}} (r \triangleright p^n)
\]

where \(p^n\) is the sequential composition of \(p\) with itself for \(n\) times and \(\bigcup\) 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 [19] that proposes a “cutting” technique restricting the model to a maximum length \(L\). In [19], 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 3.2.

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

\[
r \triangleright (\text{create } x) = r - x
\]
\[
r \triangleright (\text{forget } x) = r - x.
\]
The (qualified) function calls comply to their initial definitions in [19]:

\[
\begin{align*}
  r \triangleright (\text{call } f(l)) &= \ (r[f\cdot l]) \triangleright | f | \\
  r \triangleright (x.\text{call } f(l)) &= \ x.((x' \cdot r) \triangleright \text{call } f(x' \cdot l)).
\end{align*}
\]

Here \( f^* \) 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 \( u \).

Consider a call \( x.\text{call } f(l) \) executed on behalf of a certain client object, which applies \( f \) to a supplier object referenced by \( x \). In order to compute the aliasing induced by \( x.\text{call } f(l) \) when starting with an aliasing environment \( r \), we need to compute the aliasing induced by the execution of \( f \) from \( r \) as seen by the supplier object \( x \). This supplier view is \( x' \cdot r \) with both elements of every pair in \( r \) prefixed by the negative reference \( x' \). Intuitively, \( x' \) can be seen as a back pointer giving access to the client. In accordance, the alias calculus evaluates \( x' \cdot x \) to \( \text{Current} \). Intuitively, the negative variable \( x' \) is meant to transpose the context of the qualified call to the context of the caller.

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 \( a.\text{call } f(a) \) computes as follows:

\[
\begin{align*}
  \emptyset \triangleright a.\text{call } f(a) &= \emptyset \\
  a.(a',\emptyset) \triangleright y := a'.a &= a.(\emptyset \triangleright y := \text{Current}) = \text{dot-complete}([a.y,a]).
\end{align*}
\]

Recursive function calls can lead to infinite alias relations if they do not terminate. 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 [19]:

\[
\begin{align*}
  r \triangleright (x := s) &= \ (r_1 - x)[x = (r_1/s - x)] - ox \\
  \text{where } r_1 &= r[ox = x].
\end{align*}
\]

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

**Remark 4.** It is worth discussing the reason behind not considering transitive alias relations. Assume the following program:

\[
\text{then } x := y \text{ else } y := z \text{ end}
\]

Based on the equations (10) and (14) 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.

### 3.2. A sound over-approximation

In a non-concurrent or, 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

\[
r \triangleright p^r,
\]
Proof. The proof of (a) follows by induction on the structure of $p$.

Base case: $p ::= \text{create } x | \text{forget } x | x := s$. The result follows by Proposition 5.

Induction step. Assume $r \rightarrow p'$ is regular for all regular relations $r'$ and all basic blocks $p'$ with simpler structure than $p$.

Let $p = \text{then } p_1 \text{ else } p_2 \text{ end}$. By definition, $\tau = r \rightarrow \text{then } p_1 \text{ else } p_2 \text{ end} = (r \rightarrow p_1) \cup (r \rightarrow p_2)$. Hence, by the induction hypothesis, it follows that $\tau$ is regular.

Let $p = p_1 ; p_2$. By definition, $\tau = r \rightarrow p_1 ; p_2 = (r \rightarrow p_1) \rightarrow p_2$. By the induction hypothesis, it holds that $r \rightarrow p_1$ is regular. Hence, by the induction hypothesis, it follows that $\tau = (r \rightarrow p_1) \rightarrow p_2$ is regular as well.

Lemma 6. Let $r$ be a regular alias relation and $p$ a basic block as in (15). In a sequential setting, it holds that:

(a) $r \rightarrow p$ is regular;
(b) $r \rightarrow p^\omega$, with $n \geq 1$, is regular.

Proof. The proof of (a) follows by induction on the structure of $p$.

Base case: $p ::= \text{create } x | \text{forget } x | x := s$. The result follows by Proposition 5.

Induction step. Assume $r \rightarrow p'$ is regular for all regular relations $r'$ and all basic blocks $p'$ with simpler structure than $p$.

Let $p = \text{then } p_1 \text{ else } p_2 \text{ end}$. By definition, $\tau = r \rightarrow \text{then } p_1 \text{ else } p_2 \text{ end} = (r \rightarrow p_1) \cup (r \rightarrow p_2)$. Hence, by the induction hypothesis, it follows that $\tau$ is regular.

Let $p = p_1 ; p_2$. By definition, $\tau = r \rightarrow p_1 ; p_2 = (r \rightarrow p_1) \rightarrow p_2$. By the induction hypothesis, it holds that $r \rightarrow p_1$ is regular. Hence, by the induction hypothesis, it follows that $\tau = (r \rightarrow p_1) \rightarrow p_2$ is regular as well.
The proof of (b) is by induction on \( n \). The base case, when \( n = 1 \), follows by Lemma 6(a). The induction step follows immediately by the induction hypothesis, as \( r \triangleright p^{n+1} = (r \triangleright p^n) \triangleright p \). 

\[ \text{Lemma 7.} \quad \text{Let } r \text{ be a regular alias relation and} \]

\[ p :: = p ; p | \text{then } p \text{ else } p | \text{end } \mid \text{create } x | \text{forget } x | x := s \mid \text{loop } p \text{ end} \]  

(17)

In a sequential setting, it holds that:

(a) \( r \triangleright p \) is regular;

(b) \( r \triangleright p^n \), with \( n \geq 1 \), is regular.

**Proof.** The proof of (a) is by induction on \( n \) − the number of nested loops in \( p \).

**Base case:** \( n = 0 \). The result follows immediately by Lemma 6(a), for \( p \) a basic block as in (15).

Let \( p = p_1 ; \text{loop } q \text{ end} ; p_2 \), with \( p_1, p_2 \) and \( q \) basic blocks as in (15). It holds that

\[ r' = r \triangleright p_1 ; \text{loop } q \text{ end} ; p_2 = \bigcup_{n \in \mathbb{N}} r \triangleright p_1 \triangleright q^n \triangleright p_2. \]

Hence, by Lemma 6(b), \( r' \) is regular.

**Induction step.** Assume \( r' \triangleright p' \) is regular for all regular relations \( r' \) and \( p' \) instruction blocks with maximum \( n \) nested loops, built as in (17). Let \( p = p_1 ; \text{loop } q \text{ end} ; p_2 \), with \( p_1, p_2 \) and \( q \) instruction blocks with maximum \( n \) nested loops, built as in (17). It holds that \( r = r \triangleright p_1 ; \text{loop } q \text{ end} ; p_2 = \bigcup_{n \in \mathbb{N}} r \triangleright p_1 \triangleright q^n \triangleright p_2 \). Hence, \( r \) is regular by the induction hypothesis.

The proof of (b) is by induction on \( n \). The base case, when \( n = 1 \), follows by Lemma 7(a). The induction step follows immediately by the induction hypothesis, as \( r \triangleright p^{n+1} = (r \triangleright p^n) \triangleright p \). 

\[ \text{Lemma 8.} \quad \text{Let } r \text{ be a regular alias relation and } f(x_1, \ldots, x_n) = \{p_1 : \text{call } f(y_1, \ldots, y_n) ; p_2\} \text{ a recursive function, with } p_1, p_2 \text{ as in (17). In a sequential setting, it holds that:} \]

(a) \( r \triangleright \text{call } f(a_1, \ldots, a_n) \) is regular,

(b) \( r \triangleright \text{call } f(a_1, \ldots, a_n)^n \), with \( n \geq 1 \), is regular,

where \( a_1, \ldots, a_n \) is a valid list of actual parameters of \( f \).

**Proof.** For proving (a) we proceed as follows. First, for simplicity of notation, we write \( l_i, i \geq 0 \), in order to refer to the actual parameters of \( f \) corresponding to its \( i \)’th recursive call. Let \( l_0 = a_1, \ldots, a_n \). Then:

\[ r \triangleright \text{call } f(l_0) = r[f^* : l_0] \triangleright f \mid f \mid = r[f^* : l_0] \triangleright p_1 \triangleright \text{call } f(l_1) \triangleright p_2 = \]

\[ (r[f^* : l_0] \triangleright p_1)[f^* : l_1] \triangleright p_1 \triangleright \text{call } f(l_2) \triangleright p_2 = \ldots = \]

\[ ((r[f^* : l_0] \triangleright p_1)[f^* : l_1] \triangleright p_1 \ldots)[f^* : l_n] \triangleright p_1 \triangleright \text{call } f(l_n) \triangleright p_2 \]

Let \( r \) be the alias relation obtained from \( r \) by repeatedly replacing the actual arguments of \( f \) with their formal counterparts, in the reasoning above. Thus, also based on the results in Lemma 7, it follows that \( r \) is regular. Additionally, the aliasing information corresponding to \( r \triangleright \text{call } f(l_0) \) can be over-approximated by \( \bigcup_{n \in \mathbb{N}} r \triangleright p_1^n \triangleright p_2 \). Hence, as \( r \) is regular and \( p_1, p_2 \) are built as in (17), the statement in Lemma 8(a) is a consequence of Lemma 7(b).

The proof of Lemma 8(b) follows by induction on \( n \) and Lemma 8(a).

\[ \text{Lemma 9.} \quad \text{Consider } r \text{ a regular alias relation and a mutually recursive function } f(x_1, \ldots, x_n) = \{p_1 : \text{call } g(y_1, \ldots, y_m) ; p_2\} \text{ with } g(y_1, \ldots, y_m) = \{q_1 : \text{call } f(x_1, \ldots, x_n) ; q_2\}, \text{ and } p_1, p_2, q_1, q_2 \text{ as in (17). In a sequential setting, it holds that:} \]

\[ \]
(a) \( r \triangleright \text{call } f(a_1, \ldots, a_n) \) is regular,
(b) \( r \triangleright \text{call } f(a_1, \ldots, a_n)^n \), with \( n \geq 1 \), is regular,
where \( a_1, \ldots, a_n \) is a valid list of actual parameters of \( f \).

**Proof Sketch.** The proof follows a reasoning similar to the one in the proof of Lemma 8(a). The alias relation \( r \triangleright \text{call } f(a_1, \ldots, a_n) \) can be over-approximated by \( \bigcup_{n \in \mathbb{N}} \bar{r} \triangleright (p_1; q_1)^n \triangleright (q_2; p_2)^n \), where \( \bar{r} \) is obtained from \( r \) by repeatedly replacing the actual arguments of \( f \) and \( g \) with their formal counterparts. As \( \bar{r} \) is regular and \( p_1, p_2, q_1, q_2 \) are built as in (17), the statement in Lemma 9(a) is a consequence of Lemma 7(b).

The proof of (b) follows by induction on \( n \), and Lemma 9(a).

**Theorem 10.** Assume \( p \) a program built according to the rules in (8). Then, in a sequential setting, the relation \( \emptyset \triangleright p \) is regular.

**Proof Sketch.** The proof follows by induction on the structure of \( p \) and Lemmas 6–9.

Inspired by the idea behind the *pumping lemma for regular languages* [33], 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 infinite execution of loops or recursive function calls. We supply sound over-approximations of “lasso” relations, based on regular alias expressions.

**Definition 11 (Lassos).** Consider \( r \) and \( r' \) two alias relations, and \( x_i, y_i \), and \( z_i \) a set of possibly empty expressions, for \( i \in \{1, 2\} \). A lasso property over \( r \) and \( r' \) is defined as:

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

(18)

Intuitively, there is a lasso between \( r \) and \( r' \) whenever the elements in \( r' \) correspond to elements in \( r \) for which tails of prefixes are repeated.

**Definition 12 (Over-approximating regular aliases).** Consider \( r \) and \( r' \) such that \( \text{lasso}(r, r') = \text{true} \). Let \( x_i, y_i \), and \( z_i \) be possibly empty expressions, for \( i \in \{1, 2\} \). The set of regular alias expressions over-approximating \( r \) and \( r' \) is defined as:

\[
\text{reg}(r, r') = \{[x_1.y_1.z_1, x_2.y_2.z_2] \mid [x_1.y_1.z_1, x_2.y_2.z_2] \in r \land [x_1.y_1.z_1, x_2.y_2.z_2] \in r'\}
\]

(19)

As previously indicated, the over-approximation is sound w.r.t. the repeated application of a basic block as in (15), in the way that it does not introduce any false negatives:

**Lemma 13.** Consider \( r \) and \( r' \) two alias relations, and \( p \) a basic block as in (15). In a sequential setting, if \( r \triangleright p = r' \) and \( \text{lasso}(r, r') = \text{true} \), then the following holds for all \( n \geq 1 \):

\[
r \triangleright p^n \subseteq \text{reg}(r, r').
\]

**Proof.** We proceed by induction on \( n \).

- **Base case:** \( n = 1 \). By hypothesis it holds that \( \text{lasso}(r, r') = \text{true} \). Hence, according to the definition of \( \text{lasso}(\_, \_) \) in (18), the following holds for \( r \) and \( r' \):

\[
[x_1.y_1.z_1, x_2.y_2.z_2] \in r \iff [x_1.y_1.z_1, x_2.y_2.z_2] \in r'.
\]

Consequently, by the definition of \( \text{reg}(\_, \_) \) in (19), it is easy to see that

\[
r' \subseteq \text{reg}(r, r').
\]
• Induction step. Fix a natural number \( n \) and suppose that
\[
 r_k = r \Rightarrow p^k \subseteq \text{reg}(r, r')
\]
for all \( k \in \{1, \ldots, n\} \). We want to prove that (20) holds also for \( k = n + 1 \).
We continue by “reductio ad absurdum”. Consider
\[
r_n = r \Rightarrow p^n \subseteq \text{reg}(r, r'),
\]
and assume that
\[
r_{n+1} = r_n \Rightarrow p \not\subseteq \text{reg}(r, r')
\]
(21) Clearly, the execution of \( p \) when starting with \( r_n \) introduces an alias pair which is not in \( \text{reg}(r, r') \).
According to (12), only instructions \( \text{create} \) \( x \) and \( \text{forget} \) \( x \) that appear within \( p \), can remove alias pairs from \( r_n \). Hence, it must be the case that the regular structure of the alias information is broken via a newly added pair \([t', s']\) associated to an assignment \( x := s \) within \( p \).
Let \( p = C[x := s] \), where \( C \) is a context built according to (15), and \( x := s \) is the upper-most assignment instruction in the syntactic tree associated to \( p \) that introduces a pair \([t', s']\) \( \not\subseteq \text{reg}(r, r') \).
Assume that \( \tilde{r}_{n-1} \) and \( \tilde{r}_n \), respectively, are the intermediate alias relations obtained by reducing \( r_{n-1} \Rightarrow C[x := s] \) and \( r_n \Rightarrow C[x := s] \), respectively, according to (9)−(14), before the application of the assignment axiom corresponding to \( x := s \). As only \( \text{create} \) \( x \) and \( \text{forget} \) \( x \) can remove alias pairs from \( r_{n-1} \) and \( r_n \), respectively, and as \( x := s \) is the first assignment within \( p \), it follows that \( \tilde{r}_{n-1} \subseteq r_{n-1} \subseteq \text{reg}(r, r') \) and \( \tilde{r}_n \subseteq r_n \subseteq \text{reg}(r, r') \).
Let \( \tilde{r} \subseteq \text{reg}(r, r') \). Based on axiom (14) it follows that \( \tilde{r} \Rightarrow x := s \not\subseteq \text{reg}(r, r') \) if and only if:
\[
\begin{align*}
&\text{(i) } [x, s] \not\subseteq \text{reg}(r, r'), \text{ or, according to dot-completeness,} \\
&\text{(ii.1) } [x.a, s.a] \not\subseteq \text{reg}(r, r'), \text{ where } a \in \text{dom}(x), \text{ or,} \\
&\text{(ii.2) } [s.a, x_2.y_2.z_2] \not\subseteq \text{reg}(r, r'), \text{ where } [x.a, x_2.y_2.z_2] \in \tilde{r}.
\end{align*}
\]
Let \([t', s']\) \( \not\subseteq \text{reg}(r, r') \) be the pair introduced by \( x := s \) when starting from \( \tilde{r}_n \). Assume, based on (i) above, that \([t', s'] = [x, s] \not\subseteq \text{reg}(r, r') \). However, \([x, s] \) must have been added when handling \( r_{n-1} \Rightarrow x := s \) as well, without affecting the regularity of \( r_n \). Hence, \([x, s] \) must have been removed from \( r_n \) via \( \text{forget} \) \( x \) or \( \text{create} \) \( x \) within \( p \). Consequently, \([x, s] \) will not occur in \( \tilde{r}_n \Rightarrow C[x := s] \) either. Therefore, we conclude that all pairs \([x, s] \not\subseteq \text{reg}(r, r') \) added to \( \tilde{r}_n \) via assignments \( x := s \) within \( p \) will be eventually removed via \( \text{forget} \) \( x \) or \( \text{create} \) \( x \) within \( p \). The reasoning is similar for \([t', s'] = [x.a, s.a] \) as in (ii.1) and \([t', s'] = [s.a, x_2.y_2.z_2] \) as in (ii.2) above.
So, it follows that \( r_{n+1} = r_n \Rightarrow p \subseteq \text{reg}(r, r') \). This contradicts our assumption in (21). We conclude that \( r \Rightarrow p^n \subseteq \text{reg}(r, r') \).

\[\square\]

4. A RL-based procedure for collecting aliases

In this section we provide the specification of a RL-based mechanism collecting the alias information. The RL rules are specified using the \( K \) notation for configurations and are implemented in Maude on top of the formalization of SCOOP \[1\] (we refer to Section 5 for more details on our approach).

In short, our strategy is to start with a program built on top of the control structures in (8), then to apply their corresponding \( K \)-rules in order to get the “may aliasing” information in a designated \( K \)-cell \( (−)_\text{al} \). 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 \( K \)-like rules is finite and the aliases in the final configuration soundly over-approximate the (infinite) “may alias” relations of the calculus.
**K-like notations for configurations and (RL) rules.** K [35, 34] is an executable semantic framework based on Rewriting Logic [5]. It is suitable for defining (concurrent) languages and corresponding formal analysis tools, with implementation in K-Maude [36]. K-definitions make use of the so-called cells, which are labelled and can be nested, and (rewriting) rules describing the intended (operational) semantics. In this section we adopt a K-like notation in order to provide a RL-based specification of the alias calculus.

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

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

\[
\langle i_1 \rightarrow i_2 \ldots \rangle_k
\]

for a set of instructions to be “executed”, starting with instruction \(i_1\), followed by \(i_2\). The associative operation \(\rightarrow\) is the instruction sequencing.

A rule

\[
\langle c \ldots \rangle_n \langle c' \rangle_{n_2} \langle \ldots \rangle_{n_3}
\]

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.

We further provide the details behind the K-like specification of the alias calculus. As expected, the \(k\)-cell retains the instruction stack of the object-oriented program. We utilize cells \(\langle - \rangle_{\text{al}}\) to enclose the current alias information, and the so-called back-tracking cells \(\langle - \rangle_{\text{bkt-}}\) enabling the sound computation of aliases for the case of **then – else – 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 13.

The following K-like rules are straightforward, based on the axioms (9)-(14) in Section 3.1. Namely, the rule implementing an instruction \(p; q\) simply forces the sequential execution of \(p\) and \(q\) by positioning \(p \rightarrow q\) at the top of the continuation cell:

\[
\frac{p \rightarrow q \ldots}{\langle - \rangle_k} \hspace{2cm} (22)
\]

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:

\[
\frac{r \rightarrow x \ldots}{\langle - \rangle_k} \hspace{2cm} (23)
\]

The assignment rule restores the current alias relation according to its axiom in (14), and removes the assignment instruction from the top of the \(k\)-cell:

\[
\frac{(r_1 - t)[t = r_1/s - t] - \alpha l}{t := s \ldots \langle - \rangle_k} \hspace{2cm} \text{with } r_1 = r[\alpha t = t] \hspace{2cm} (24)
\]

Handling **then\( p\) else \(q\) end** statements is more sophisticated, as it requires instrumenting a stack-based mechanism enabling the computation of the union of alias relations \(r \rightarrow p \cup r \rightarrow q\) in three steps. First, we define the K-like rule:

\[
\frac{r \rightarrow \langle - \rangle_{\text{bkt-te}}}{\langle - \rangle_k} \hspace{2cm} (25)
\]

saving at the top of the back-tracking stack \(\langle - \rangle_{\text{bkt-te}}\) the initial alias relation \(r\) to be modified by both \(p\) and \(q\), via two cells \(\langle r, p \rangle_t\) and \(\langle r, q \rangle_e\), respectively. Note that the original instruction in the \(k\)-cell is replaced by
a construction marking the end of the executions corresponding to the then and else branches with \texttt{et} and \texttt{ce}, respectively.

Second, whenever the successful execution of \( p \) (signaled by \texttt{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 \( \langle - \rangle_{al} \), and by positioning \texttt{ce} at the top of the \( k \)-cell. The new alias information after \( p \), denoted by \( \langle r', p \rangle_t \), is updated in the back-tracking cell:

\[
\langle r', p \rangle_t = \langle \texttt{et} q \texttt{ce} \ldots \rangle_k \langle \langle r, p \rangle_t, \langle r, q \rangle_u \ldots \rangle_{bkt-te}
\]

(26)

Eventually, if the successful execution of \( q \) (marked by \texttt{ce} at the top of \( \langle - \rangle_k \)) 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 (26). The corresponding back-tracking information is removed from \( \langle - \rangle_{bkt-te} \), and the next program instruction is enabled in the \( k \)-cell:

\[
\langle r'', \texttt{loop} p \texttt{end} \ldots \rangle_{al-o} \langle \langle r', p \rangle_{al-o} \langle r, q \rangle_u \ldots \rangle_{bkt-te}
\]

(27)

For \texttt{loop} \( p \) \texttt{end}, we utilize a meta-construction \( p \downarrow \texttt{loop} p \texttt{end} \) simulating the set union in (11), and a back-tracking stack \( \langle - \rangle_{bkt-te} \) collecting the alias information obtained after each execution of \( p \). Moreover, the K-like specification exploits the result in Lemma 13. 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 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 \( \langle r \rangle_{al-o}(p) \):

\[
\langle r \rangle_{al-o}(p) = \langle \texttt{loop} p \texttt{end} \ldots \rangle_k \langle \langle r \rangle_{al-o}(p) \rangle_{bkt-l}
\]

(28)

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

\[
\langle r' \rangle_{al-o}(p) = \langle \texttt{loop} p \texttt{end} \ldots \rangle_k \langle \langle r \rangle_{al-o}(p) \rangle_{bkt-l}
\]

(29)

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 \( \langle - \rangle_{bkt-te} \) and the \texttt{loop} instruction is eliminated from the \( k \)-cell:

\[
\langle r' \rangle_{al-o}(p) = \langle \texttt{loop} p \texttt{end} \ldots \rangle_k \langle \langle r \rangle_{al-o}(p) \rangle_{bkt-l}
\]

(30)

For handling function calls such as \texttt{call} \( f(l) \) we use a meta-construction \( \mid f \mid \) \texttt{call} \( f(l) \). Here \( \mid f \mid \) stands for the body of \( f \) and \( \texttt{call} \) marks the end of the corresponding execution. Moreover, a stack \( \langle - \rangle_{bkt-cf} \) 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 \( \texttt{call} \) and \( \langle - \rangle_{bkt-cf} \) are parameterized by \( f \) – the name of the function. An example illustrating this reasoning mechanism is provided in Section 4.1.

The first K-like rule for handling function calls matches the associated axiom in (13): the alias information is set to \( r[f^{*}l] \), whereas the next instructions to be executed are given by \( \mid f \mid \). Note that the original aliasing is retained in the (initially empty) back-tracking cell via \( \langle r \rangle_{al-o} \).

\[
\langle r \rangle_{al-o}(p) = \langle \texttt{call} f(l) \ldots \rangle_k \langle \langle r \rangle_{al-o} \rangle_{bkt-cf}
\]

(31)
Remark 14. 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 $r[\ell]^*$.

A successful execution of $\text{call } f(l)$ is distinguished by the occurrence of $\llbracket f \rrbracket$ 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:

$$\langle r' \rangle_{\text{al}} \langle \llbracket f \rrbracket \ldots \rangle_k \langle \langle r \rangle_{\text{al-o}} \ldots \rangle_{\text{bkt-cf}}$$

(32)

Recursive calls are treated by means of two rules. Note that a recursive context is identified whenever the current program instruction is of shape $\text{call } f(l)$ and the associated back-tracking structure is not empty, i.e., rule (31) 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 13, and by removing $\text{call } f(l)$ from the $k$-cell:

$$\langle \langle r' \rangle_{\text{al}} \langle \llbracket f(l) \rrbracket \ldots \rangle_k \langle \langle r \rangle_{\text{al-o}} \ldots \rangle_{\text{bkt-cf}} \rangle_{\text{bkt-cf}}$$

if $\text{lasso}(r, r')$ (33)

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:

$$\langle \langle r' \rangle_{\text{al}} \langle \llbracket f(l) \rrbracket \ldots \rangle_k \langle \langle r \rangle_{\text{al-o}} \ldots \rangle_{\text{bkt-cf}} \rangle_{\text{bkt-cf}}$$

if not $\text{lasso}(r, r')$ (34)

Qualified calls $x.\text{call } f(l)$ are handled by two rules as follows. First, based on the definition in (13), 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 $\llbracket qf \rrbracket$ is utilized in order to mark the end of the qualified call in the continuation cell, similarly to the rule (31). The caller is stored in a back-tracking stack $\langle \langle x \rangle_{\text{bkt-qf}} \rangle_{\text{bkt-qf}}$ also parameterized by $f$ – the name of the function. The current instruction in the $k$-cell becomes $\text{call } f(x'.l)$, as expected:

$$\langle \langle r \rangle_{\text{al}} \langle \llbracket x.\text{call } f(l) \rrbracket \ldots \rangle_k \langle \langle r \rangle_{\text{al-o}} \ldots \rangle_{\text{bkt-qf}} \rangle_{\text{bkt-qf}}$$

(35)

Second, when the successful termination of the qualified call is signaled by $\llbracket qf \rrbracket$ 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 $\llbracket qf \rrbracket$:

$$\langle \langle r \rangle_{\text{al}} \langle \llbracket qf \rrbracket \ldots \rangle_k \langle \langle x \rangle_{\text{al-o}} \ldots \rangle_{\text{bkt-qf}} \rangle_{\text{bkt-qf}}$$

(36)

In a non-concurrent setting, the machinery orchestrating the K-like rules introduced in this section implements an algorithm that always terminates and provides a sound over-approximation of “may aliasing”.

Theorem 15. Consider $p$ a program built on top of the control structures in (8), that executes in a sequential setting. Then, the application of the corresponding K-like rules when starting with $p$ and an empty alias relation, is a finite rewriting of shape

$$\langle \langle \emptyset \rangle_{\text{al}} \langle p \rangle_k \rangle_{\text{al}} \langle \langle r \rangle_{\text{al}} \langle \ldots \rangle_k \rangle_{\text{al}} \xrightarrow{(s)} \langle \langle r \rangle_{\text{al}} \langle \ldots \rangle_k \rangle_{\text{al}},$$

with $r$ a sound over-approximation of the aliasing information corresponding to the execution of $p$, and $(s)$ the rewrite relation defined by the rules (22)–(36).
Proof Sketch. First, note that the rewrite system in (22)–(36) is confluent, as there is only one way of performing the rewritings starting with program p, by matching at top (and at the top of the cells ⟨−⟩bkt-cf). Then, observe 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 K-like 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 13.

Observe that the RL-based machinery can simulate precisely the “cutting at length L” technique in [19]. It suffices to disable the rules (♣) and stop the rewriting after L steps.

The naturalness of applying the resulted aliasing framework is illustrated in the example in Section 4.1, for the case of two mutually recursive functions.

4.1. The K-machinery by example

For an example, in this section we show how the K-machinery developed in Section 4 can be used in order to extract the alias information for the case of two mutually recursive functions defined as:

\[
\begin{align*}
f(x) \{ x := x.a; \text{call } g(x) \} \\
g(x) \{ x := x.b; \text{call } f(x) \}
\end{align*}
\]

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

The execution of \texttt{call } f(x), when starting with an empty alias relation \( r \), produces the alias expressions:

\[
\left[ x, x.(a.b)^* \right] \quad \left[ x.a, x.(a.b)^*.a \right] \quad \left[ x.b, x.(a.b)^*.b \right]
\]

(37)

The associated rewriting is depicted in Figure 4.1. The whole procedure starts with an empty alias relation \( r = \emptyset \), and \texttt{call } f(x) in the continuation stack. Then, the corresponding 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 \( \boxed{\square} f \boxed{\square} f \) — in the k-cell. This triggers the application of rule (33) enabling the “regular” over-approximation as in Lemma 13.

Our example also illustrates the importance of isolating the back-traced alias information in cells of shape \( ⟨−⟩_{bkt-cf} \) parameterized by the (possibly recursive) function \( f \). More explicitly, rule (33) 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 \( ⟨ ⟨ r_2 \rangle_{bkt-cf} \ldots⟩ \) associated to the previous executions of \( f(l) \).

As introduced in (18), an alias relation \( r' \) is a lasso of a relation \( r \) whenever there is a one-to-one correspondence between their elements as follows:

\[
\left[ x_1 y_1 z_1, x_2 y_2 z_2 \right] \in r \iff \left[ x_1 y_1 z_1, x_2 y_2 z_2 \right] \in r'.
\]

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 (33), 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:

<table>
<thead>
<tr>
<th>[ x_1 y_1 z_1, x_2 y_2 z_2 ] \in r_2</th>
<th>[ x_1 y_1 z_1, x_2 y_2 z_2 ] \in r_4</th>
<th>y_1</th>
<th>z_1</th>
<th>y_2</th>
<th>z_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>x.e</td>
<td>e</td>
<td>x.a.b</td>
<td>e</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x.e</td>
<td>a</td>
<td>x.b</td>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x.e</td>
<td>b</td>
<td>x.a.b</td>
<td>b</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Here \( \varepsilon \) stands for the empty alias expression.

Moreover, according to rule (33), the lasso shaped by \( r_2 \) and \( r_4 \) also causes the (otherwise infinite) recursive calls to stop, as \( \text{call} f(l) \) 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 13), defined precisely as in (37).

\[
\begin{align*}
\langle r \rangle_{\text{al}} & \langle \text{call} f(x) \rangle_k \\
\langle \cdot \rangle_{\text{bkt-cf}} & \langle \cdot \rangle_{\text{bkt-cg}} \\
\Downarrow & \ (31) \\
\langle r \rangle_{\text{al}} & \langle x := x.a; \text{call} g(x) \rangle_k \\
\langle \cdot \rangle_{\text{al-o}} & \langle \cdot \rangle_{\text{bkt-cg}} \\
\Downarrow & \ (24) \\
\langle r_1 \rangle_{\text{al}} & \langle \text{call} g(x) \rangle_k \\
\langle \cdot \rangle_{\text{al-o}} & \langle \cdot \rangle_{\text{bkt-cg}} \\
\text{where } r_1 & = \{ [x, x.a], [x.a, x.a.a], [x.b, x.a.b] \} \\
\Downarrow & \ (34) \\
\langle r_1 \rangle_{\text{al}} & \langle x := x.b; \text{call} f(x) \rangle_k \\
\langle \cdot \rangle_{\text{al-o}} & \langle \cdot \rangle_{\text{bkt-cg}} \\
\Downarrow & \ (24) \\
\langle r_2 \rangle_{\text{al}} & \langle \text{call} f(x) \rangle_k \\
\langle \cdot \rangle_{\text{al-o}} & \langle \cdot \rangle_{\text{bkt-cg}} \\
\text{where } r_2 & = \{ [x, x.a.b], [x.a, x.a.b.a], [x.b, x.a.b.b] \} \\
\Downarrow & \ (34) \\
\langle r_2 \rangle_{\text{al}} & \langle x := x.a; \text{call} g(x) \rangle_k \\
\langle \cdot \rangle_{\text{al-o}} & \langle \cdot \rangle_{\text{al-o}} \\
\Downarrow & \ (24) \\
\langle r_3 \rangle_{\text{al}} & \langle \text{call} g(x) \rangle_k \\
\langle \cdot \rangle_{\text{al-o}} & \langle \cdot \rangle_{\text{bkt-cg}} \\
\text{where } r_3 & = \{ [x, x.a.b.a], [x.a, x.a.b.a.a], [x.b, x.a.b.a.b] \} \\
\Downarrow & \ (34) \\
\langle r_3 \rangle_{\text{al}} & \langle x := x.b; \text{call} f(x) \rangle_k \\
\langle \cdot \rangle_{\text{al-o}} & \langle \cdot \rangle_{\text{al-o}} \\
\Downarrow & \ (24) \\
\langle r_4 \rangle_{\text{al}} & \langle \text{call} f(x) \rangle_k \\
\langle \cdot \rangle_{\text{al-o}} & \langle \cdot \rangle_{\text{al-o}} \\
\text{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 & \ (33) \\
\langle \text{reg}(r_2, r_4) \rangle_{\text{al}} & \langle \cdot \rangle_k \\
\langle \cdot \rangle_{\text{al-o}} & \langle \cdot \rangle_{\text{bkt-cg}} \\
\Downarrow & \ (32) \\
\langle \{ [x, x.(a.b)^*], [x,a,x.(a.b)\ast.a], [x.b, x.(a.b)\ast.b] \} \rangle_{\text{al}} & \langle \cdot \rangle_k \langle \cdot \rangle_{\text{bkt-cf}} \langle \cdot \rangle_{\text{bkt-cg}} \\
\Downarrow & \ (*)(32)
\end{align*}
\]

Figure 1: Over-approximating aliasing.
5. Aliasing in SCOOP

In this section we provide a brief overview on the integration and applicability of the alias calculus in SCOOP. First, recall from Section 2 that the Maude semantics of SCOOP in [1] is defined over tuples of shape

\[ \langle p_1 :: St_1 | \ldots | p_n :: St_n, \sigma \rangle \]

where, \( p_i \) and \( St_i \) stand for processors and their call stacks, respectively. \( \sigma \) is the state of the system and it holds information about the heap and the store.

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

\[ a \text{ is fresh} \]

\[ \Gamma \vdash \langle p :: t := s; St, \sigma \rangle \rightarrow \langle p :: \text{eval}(a, s); \text{wait}(a); \text{write}(t, a.data); St, \sigma \rangle \]  

(38)

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 \( \Gamma \) is a typing environment [37] containing the class hierarchy of a program and all the type definitions.

The K-like rule for assignments

\[ \land \rightarrow \langle r_1 \rangle \text{al}(r_1 - t)\[ t := s \ldots \rangle \kappa \]  

(24)

can be straightforwardly integrated in (38) by enriching the SCOOP configuration with a new component alias encapsulating the alias information, and considering instead the transition:

\[ \Gamma \vdash \langle p :: t := s; St, \sigma, \text{alias}_{\text{old}} \rangle \rightarrow \langle p :: \text{eval}(a, s); \text{wait}(a); \text{write}(t, a.data); St, \sigma, \text{alias}_{\text{new}} \rangle \]

where

\[ \text{alias}_{\text{old}} = r \quad \text{alias}_{\text{new}} = (r_1 - t)[t = (r_1/s - t)] - ot \]

with \( r \) and \( r_1 \) as in (24). The integration of all the K-rules of the alias calculus on top of the Maude formalization of SCOOP is achieved by following a similar approach.

Nevertheless, it is important to point out that in order of the sound over-approximating aliases to be constructed as in Section 4, Theorem 15, one needs to simulate a SCOOP sequential setting. This is possible in SCOOP by creating all the objects in the program to be analysed as objects handled by the same processor. An example computing the aliasing information is provided at:

https://dl.dropboxusercontent.com/u/1356725/SCOOP-SCP.zip

Simply run the command

\> maude SCOOP.maude ..\examples\linked_list-test.maude

for five iterations of the loop. As can be observed based on the code in linked_list-test.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 field \( \text{next} \) and setter \( \text{set_next} \) is declared as:

```maude
class 'NODE
  create {'make} (  
    attribute {'ANY} 'next : [?, . , 'NODE] ;  
    procedure {'ANY} 'set_next ( 'a_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 “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, aliasing-on).

Observe that the flag for performing the alias analysis is switched to “on”.

The relevant parts of the corresponding Maude output after executing the aforementioned command are as follows:

rewrite in SYSTEM :
[...] settings('LINKED_LIST_TEST, 'make, aliasing-on))

|-
{0}proc(0) :: nil | {0}proc(1) :: nil,
{{'x0 ; 'x0}} U {{'x0  ; 'x1.'next}} U
{{'x0 ; 'x2.'next.'next}} U {{'x0 ; 'x3.'next.'next.'next}} U
{{'x0  ; 'x4.'next.'next.'next.'next}} U
[...]
{{'x3 ; 'x3}} U {{'x3 ; 'x4.'next}}

state
  heap [...]
  store [...]
end

In short, one can see that two processors that were created finished executing the instructions on their corresponding call stacks: {0}proc(0) :: nil and {0}proc(1) :: nil. The aliased expressions include, as expected based on (7), pairs of shape \([x_i; x_{i+k}. next^k]\). Moreover, the output displays the contents of the current system state, by providing information on the heap and store, as formalized in [1].

6. Deadlocking in SCOOP

In what follows we provide the details behind the formalization and the implementation of a deadlock detection mechanism for SCOOP. We discuss how Maude can be exploited in order to test and, respectively, model-check deadlocks in the context of SCOOP programs, we analyze the associated challenges and propose a series of corresponding solutions.

6.1. Formalizing deadlocks in SCOOP

Recall that the key idea of SCOOP is to associate to each object declared as separate a processor, or handler in charge of executing the routines of that object. Assume a processor \( p \) that performs a call \( o.f(a_1, a_2, \ldots) \) on a separate object \( o \), with separate arguments \( a_i \) (\( i \geq 1 \)). As previously stated in Section 2, according to the SCOOP semantics, the application of the call \( f(\ldots) \) will wait until it has been able to lock all the separate objects associated to \( a_1, a_2, \ldots \). This reservation mechanism enables deadlocks to occur whenever a set of processors reserve each other circularly.

Definition 16 (Deadlock). For a processor \( p \), let \( W(p) \) be the set of handlers \( p \) waits for its asynchronous execution, and \( H(p) \) represent the set of resources \( p \) already acquired. A deadlock exists if for some set \( D \) of processors the following holds:

\[
(\forall p \in D).(\exists p' \in D).(p \neq p') \land (W(p) \cap H(p') \neq \emptyset). \quad (39)
\]
A deadlock can happen, for instance, in a Dining Philosophers scenario, where both philosophers and forks are objects residing on their own processors. Consider, for an example, two separate objects $p_1$ and $p_2$ denoting two philosophers, and two separate objects $f_1$ and $f_2$ denoting two forks. Assume that $p_1$ picks $f_1$, whereas $p_2$ picks $f_2$. Then, the following hold whenever $p_1$ and $p_2$ want to acquire both forks:

$$H(p_1) = \{f_1\} = W(p_2)$$

and

$$H(p_2) = \{f_2\} = W(p_1).$$

Hence, (16) is satisfied for $D = \{p_1, p_2\}$.

The integration of a deadlock detection mechanism based on Definition 16 on top of the SCOOP formalization in [1] is immediate. As already presented in Section 2, the operational semantics of SCOOP is defined over tuples of shape:

$$⟨p_1::St_1 | \ldots | p_n::St_n, σ⟩$$

where, $p_i$ and $St_i$ stand for processors and their call stacks, respectively, and $σ$ is the state of the system. Given a processor $p'$ as in (39), the set $H(p')$ corresponds, based on [1], to $σ.rq \_locks(p')$. Whenever the top of the instruction stack of a processor $p$ is of shape lock($\{q_1, \ldots, q_n\}$), we say that the wait set $W(p)$ is the set of processors $\{q_1, \ldots, q_n\}$. Hence, assuming a predefined system configuration $⟨deadlock⟩$, the SCOOP transition rule corresponding to (39) can be written as:

$$\frac{(\exists D \subseteq σ.procs). (\forall p \in D). (\exists p' \in D). (p \neq p') \wedge (\forall p \in D). (\exists p' \in D). (p \neq p') \wedge (\sigma.rq \_locks(p').has(q_i))}{(aqs, σ) → (deadlock)}$$

Note that $σ.procs$ in (40) returns the set of processors in the system, whereas $aqs$ stands for the list of these processors and their associated instruction stacks (separated by the associative and commutative operator “|”). We use “…” to represent an arbitrary sequence of processors and processor stacks.

6.2. Testing deadlocks

We implemented (40) and tested the deadlock detection mechanism on top of the formalization in [1] for the Dining Philosophers problem. A case study considering two philosophers sharing two forks can be run by downloading the SCOOP formalization at:

https://dl.dropboxusercontent.com/u/1356725/SCOOP-SCP-deadlock.zip

and executing the command

> maude SCOOP.maude ..\examples\dining-philosophers-rewrite.maude

The class implementing the philosopher concept is briefly given below:

```maude
class 'PHILOSOPHER
create { 'make } (  
  attribute { 'ANY } 'left : [!,T,'FORK] ;  
  attribute { 'ANY } 'right : [!,T,'FORK] ;

  procedure { 'ANY } 'make ( 'fl : [!,T,'FORK] ;  
    'fr : [!,T,'FORK] ; )
    do ( assign ('left, 'fl) ; assign ('right, 'fr) ; )
    [\ldots]
  end ;

  procedure { 'ANY } 'eat_wrong (nil)
    do ( command ('Current . 'pick_in_turn('left ;)) ; )
    [\ldots]
  end ;

  procedure { 'ANY } 'pick_in_turn ('f : [!,T,'FORK] ; )
    do ( command ('Current . 'pick_two('f ; 'right ;)) ; )
    [\ldots]
  end ;
19```
procedure { 'ANY } 'pick_two ('fa : [!,T,’FORK] ; 'fb : [!,T,’FORK] ; )
do ( command ('fa . 'use(nil)) ; command ('fb . 'use(nil)) ; )
end ;

It declares two forks – 'left and 'right of type 'FORK, that can be handled by any processor (T) and that cannot be Void (!). Assume two philosophers p1 and p2 (of separate type PHILOSOPHER) and two forks f1 and f2 (of separate type FORK). Moreover, assume that 'left and 'right for p1 correspond to 'f1 and 'f2. For the case of p2 they correspond to 'f2 and 'f1, respectively. Asynchronously, p1 and p2 can execute eat_wrong, which calls pick_in_turn(left). In the context of p1, the actual value of left is f1, whereas for p2 is f2. Consequently, both resources f1 and f2, respectively, may be locked “at the same time” by p1 and p2, respectively. Note that pick_in_turn subsequently calls pick_two that, intuitively, should enable the philosophers to use both forks. Thus, if f1 and f2, respectively, are locked by p1 and p2, respectively, the calls pick_two(f2, f1) and pick_two(f1, f2) corresponding to p1 and p2 will (circularly) wait for each other to finish. According to the SCOOP semantics, pick_two(f1, f2) is waiting for p2 to release f2, whereas pick_two(f2, f1) is waiting for p1 to release f1, as the forks are passed to pick_two(...) as separate types. In the context of SCOOP, this corresponds to a Coffman deadlock [38].

The entry point of the program implementing the Dining Philosophers example is the function 'make in the class APPLICATION, which asks the two philosophers p1 and p2 to adopt a wrong eating strategy as above, possibly leading to a deadlock situation. The flag enabling the deadlock analysis as in (40) is set to “on”. This information is specified using a Dining Philosophers configuration (DP-config) specified by settings('APPLICATION, 'make, deadlock-on).

Unfortunately, none of the executions of the Dining Philosophers scenario by simply invoking the Maude rewrite command lead to a deadlock situation. Each of our tests displayed the output:

rewrite in SYSTEM :
[...]
settings('APPLICATION, 'make, deadlock-on)

| |- {0}proc(0) :: nil | {0}proc(1) :: nil | {0}proc(3) :: nil
| |- {0}proc(5) :: nil | {0}proc(7) :: nil
| {-}proc(9) :: nil | {-}proc(11) :: nil

consisting of a list of processors (including the handlers of both the philosophers and the forks) with empty call stacks (: :: nil). This indicates that every time, the two philosophers proceeded by lifting their forks simultaneously, hence no deadlock was possible.

One possible workaround is to use predefined strategies [39] in order to guide the rewriting of the Maude rules formalizing SCOOP towards a ⟨deadlock⟩ system configuration. An example of applying such a strategy for the Dining Philosophers case can be tested by running the command:

> maude SCOOP.maude ..\examples\dining-philosophers-strategy.maude

The command srew [...] using init ; parallelism(lock) ; [...] ; deadlock-on forces the execution of a pick_in_turn approach as above. This determines Maude to first trigger the rule [init] in the SCOOP formalization in [1]. This makes all the required initializations of the bootstrap processor. Then, one of the processors that managed to lock the necessary resources is (“randomly”) enabled to proceed to the asynchronous execution of its instruction stack, according to the strategy parallelism(lock). The last step of the strategy calls the rule [deadlock-on] implementing the Coffman deadlock detection as in (40).

This time the guided rewriting leads, indeed, to one solution identifying a deadlock. The relevant parts of the corresponding Maude output are as follows:

srewrite in SYSTEM :
[...] settings('APPLICATION, 'make, deadlock-on)
using init ; parallelism{lock} ; [...] ; deadlock-on.

Solution 1
result Configuration: deadlock

Nevertheless, such an approach requires lots of ingeniousness (our strategy has more than 300 rules!) and, moreover, is not automated.

6.3. Model-checking deadlocks

In this section we provide an overview on our approach to model-checking deadlocks for SCOOP, using the LTL Maude model-checker [40]. As mentioned in the beginning of the current paper, the idea behind our work is to exploit the unifying flavor of the Maude executable semantics of SCOOP [1]. The latter integrates both the formalization of the language and its concurrency mechanisms, thus enabling using the semantic framework for program analysis purposes, “for free”.

One possible way to proceed is by simply running the Maude LTL model-checker on a Dining Philosophers configuration (DP-config) as thoroughly discussed in Section 6.2:

red modelCheck(DP-config, [ ] no-deadlock-mck).

Intuitively, [ ] no-deadlock-mck is a safety property stating the freedom from deadlocks. The predicate witnessing the absence of deadlocks is defined as:

\[
eq < \text{deadlock} > \models \text{no-deadlock-mck} = \text{false}.
eq \text{sys::Configuration} \models \text{no-deadlock-mck} = \text{true} \text{ [wise]}.\]

Above, \( \langle \text{deadlock} \rangle \) is the predefined system configuration “signalling” deadlocks in the SCOOP rule (40), whereas \( \text{sys::Configuration} \) is an arbitrary SCOOP system configuration.

Unfortunately, running the LTL model checker led to the state explosion problem. At a first look, the issue was caused by the size of the SCOOP formalization in [1] which includes all the aspects of a real concurrency model.

As a first step, we reduced this formalization by eliminating the parts that are not relevant in the context of deadlocking; examples include the garbage collection and the exception handling mechanisms.

In addition, we provided a simplified, abstract semantics of SCOOP based on aliasing. This idea stems from the fact that SCOOP processors are known from object references, that may be aliased. Therefore, the SCOOP semantics can be simplified by retaining within the corresponding transition rules only the information important for aliasing. Consider, for instance, the rules (2)–(4) specifying “if” instructions in Section 2. The abstract transition rule omits the evaluation of the conditional and computes the aliasing information similarly to the semantics of then...else...end in (10), in Section 3. The abstraction collects the aliases resulted after the execution of both “if” and “else” branches:

\[
\langle p :: \text{if } e \text{ then } St_1 \text{ else } St_2 \text{ end}; St, \sigma, alias_{old} \rangle \rightarrow \langle p :: St, \sigma, alias_{new} \rangle
\] (41)

Observe that the SCOOP system configurations in (1) are enriched with a new component alias consisting of a set of alias expressions. Above, \( alias_{old} \) is the aliasing before the execution of the “if” instruction, whereas, intuitively, \( alias_{new} \) stands for \( alias_{old} \cup alias_{old} \rightarrow St_2 \).

As a second step, we analyzed the implementation in [1] from a more engineering perspective, and identified a series of design decisions that either slowed down considerably the rewriting or made the search space grow unnecessarily large.

After running some experiments, we understood that the parallelism rule

\[
\langle p_1 :: St_1, \sigma \rangle \rightarrow \langle p'_1 :: St'_1, \sigma' \rangle
\]
\[
\langle p_1 :: St_1 | p_2 :: St_2, \sigma \rangle \rightarrow \langle p'_1 :: St'_1 | p_2 :: St_2, \sigma' \rangle
\] (42)
in [1] was increasing the rewriting time. Though elegant from the formalization perspective, the usage of this rule was not efficient. Therefore, we eliminated it from the SCOOP semantics and made the remaining rules apply directly, by matching at top. For instance, the abstract rule (41) formalizing “if” instructions in the context of one processor $p$ becomes:

$$
\langle p::\text{if } e \text{ then } St_1 \text{ else } St_2 \text{ end} : St \mid aqs, \sigma, \text{alias}_{old} \rangle \rightarrow
\langle p::St \mid aqs, \sigma, \text{alias}_{new} \rangle
$$

for an arbitrary list $aqs$ of processors and their instruction stacks. For Dining Philosophers, for example, this modification reduced the rewriting time from around 10s to less than 1s.

Recall that SCOOP processors communicate via channels. The implementation in [1] creates fresh channels (as in (2), for instance) parameterized by natural indexes. This was inefficient for model-checking purposes, as the state space contained many identical states up-to channel naming.

The implementation of the above observations enabled us to successfully identify a deadlock situation in a Dining Philosophers scenario, by using the Maude LTL model-checker. The new (reduced) formalization of SCOOP can be found at:


This approach successfully identifies a SCOOP deadlock configuration:

```plaintext
[...] proc(7) ::
lock({proc(3)} U {proc(5)});
command('fa . 'use(nil)) ; command('fb . 'use(nil));
[...]
proc(9) ::
lock({proc(3)} U {proc(5)});
command('fa . 'use(nil)) ; command('fb . 'use(nil));
[...]

state
regions(
   proc(3) --> {ref(4)} # proc(5) --> {ref(6)} #
   proc(7) --> {ref(8)} # proc(9) --> {ref(10)} #
[...] )
heap
objects
   'f1 --> ref(4) # 'f2 --> ref(6) #
   'p1 --> ref(8) # 'p2 --> ref(10) #
[...]
end
```

At a closer look, the information within the SCOOP deadlock state above identifies two processors $\text{proc}(7)$ and $\text{proc}(9)$ that cannot process the $\text{lock}({\{\text{proc}(3)} \cup \{\text{proc}(5)\}})$ statement at the top of their corresponding instruction stacks. Observe, based on the content of $\text{regions}$ above, that $\text{proc}(7)$ is the processor handling the reference $\text{ref}(8)$ which corresponds to the philosopher object $'p1$. Similarly, $\text{proc}(9)$ corresponds to the philosopher $'p2$, whereas $\text{proc}(3)$ and $\text{proc}(5)$ are associated to the fork objects $'f1$ and $'f2$, respectively. Hence, it must be the case that each of the philosophers asynchronously picked (i.e., locked) one of the two forks by executing $\text{pick\_in\_turn(...)}$. This way, locking both fork resources ($\text{lock}({\{\text{proc}(3)} \cup \{\text{proc}(5)\}})$) by any of the philosophers, in order to execute $\text{pick\_two(...)}$, is impossible according to the SCOOP semantics of locks.

Unfortunately, model-checking deadlocks in a scenario with five philosophers, for instance, takes unacceptably long time. Further improvements may be obtained by following the same recipe of collapsing
semantically equivalent states, from the deadlocking perspective. A major source of redundancy is represented by the so-called regions in [1] that, intuitively, manage all the objects handled by the same processor. Their elimination from the SCOOP abstract state could improve the overall time performance by enabling the model-checker to make less identifications.

An additional observation is that most of the rewrite rules formalizing SCOOP are conditional and have rewrites in their conditions. This makes their execution expensive and inefficient for model-checking. A reduction semantics of SCOOP [30, 41] is therefore worth investigating as future work.

7. Conclusions and pointers to future work

The focus of this paper is on building a toolbox for the analysis of SCOOP programs, with emphasis on an alias analyzer and a deadlock detector. The naturalness of our approach consists in exploiting the recent formalization of SCOOP in [1], that is executable and implemented in Maude [6]. This provides a unifying framework that can be used not only to reason about the SCOOP model and its design as in [1], but also to analyze SCOOP programs via Maude’s analysis tools.

Of particular interest for the aliasing tool is the calculus introduced in [19], which abstracts the aliasing information in terms of explicit access paths referred to as “alias expressions”. We provide an extension of this calculus from finite alias relations to infinite ones corresponding to loops and recursive calls. Moreover, we devise an associated RL-based executable specification using K notation [34]. In Theorem 15 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 alias expressions in terms of regular expressions, as in Lemma 13. The integration of the alias calculus on top of the Maude formalization of SCOOP [1] is straightforward, based on the aforementioned executable specification of the calculus. Executions of SCOOP programs can be simulated by simply exploiting the Maude rewriting capabilities, hence the computation of the corresponding aliasing information is immediate.

We agree that it could be worth presenting our analysis as an abstract interpretation (AI) [43]. A modeling exploiting the machinery of AI (based on abstract domains, abstraction and concretization functions, Galois connections, fixed-points, etc.) is an interesting topic left for future investigation.

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.

Another research direction is to derive alias-based abstractions for analyzing concurrent programs. We foresee possible connections with the work in [44] 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 [19] can be exploited in order to derive finite 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 [45] that handles point-to graphs via a stack-based algorithm for fixed-point computations.

Related to deadlock detection, the second contribution of this paper, we provided a formalization based on sets of acquired resources and sets of handlers processors wait to lock in their attempt to execute asynchronously. This definition corresponds to Coffman deadlocks [38] in the context of SCOOP, occurring whenever there is a set of processors reserving each other circularly. We introduced the equivalent SCOOP semantic rule and discussed the results of using Maude in order to analyze deadlocks in the context of a Dining
Philosophers scenario. On the one hand, the SCOOP semantics in [1] is very large, as it incorporates all the aspects of a real concurrency model. On the other hand, the formalization in [1] was tailored for reasoning about the SCOOP model, and not necessarily about SCOOP applications, hence it includes design decisions (such as index-based naming of communication channels) that make the state space grow unacceptably large for model-checking purposes. As a workaround for the model-checking issue, we presented the idea behind building an abstract semantics of SCOOP based on aliases, together with a series of implementation improvements that eventually enabled the Maude LTL model-checker to correctly identify deadlocks. A survey on abstracting techniques on top of Maude executable semantics is provided in [5].

We leave investigating the relationship between the original SCOOP semantics and the aliasing-based one as future work. At this point we observe that the abstract alias semantics introduces false positives (w.r.t. deadlocking as well) due to the over-approximating nature of the alias calculus.

As a clear direction for future work we consider designing and analyzing deadlock situations for more SCOOP applications. Based on the experience so far, this would help better understand and observe the SCOOP state space, thus providing hints for further improvements in the context of model-checking, for instance. As we could already see, major advances in this regard are obtained when semantically equivalent states are identified and collapsed within the same equivalence classes. This was the case of the indexed-based communication channels in Section 6.3. Similar redundant states are introduced by the so-called “regions” in SCOOP “administrating” objects handled by the same processor. Nevertheless, as the rewrite rules defining the semantics of SCOOP are conditional and have rewrites in their conditions, model-checking can still be expensive. A reduction semantics [30, 41] of SCOOP is therefore worth investigating.

Acknowledgements. We are grateful for valuable comments to the anonymous reviewers of SCP, FTSCS’14, Măriuca Asăvoae, Alexander Kogtenkov, José Meseguer, Benjamin Morandi and Sergey Velder.

The research leading to these results has received funding from the European Research Council under the European Union’s Seventh Framework Programme (FP7/2007-2013) / ERC Grant agreement no. 291389.

References

URL http://dx.doi.org/10.3233/978-1-60750-065-0-7
URL http://dx.doi.org/10.1109/IROS.2014.6942763
URL http://dx.doi.org/10.1016/j.ic.2013.08.004
URL http://doi.acm.org/10.1145/99583.99599
URL http://doi.acm.org/10.1145/161494.161501
URL http://doi.acm.org/10.1145/567532.567556