Storeless Semantics and Alias Logic

Marius Bozga
Marius.Bozga@imag.fr
VERIMAG
2, Avenue de Vignate
38610 GiŁres France

Radu Iosif
Radu.Iosif@imag.fr
VERIMAG
2, Avenue de Vignate
38610 GiŁres France

Yassine Laknech
Yassine.Lakhnech@imag.fr
VERIMAG
2, Avenue de Vignate
38610 GiŁres France

ABSTRACT

Pioneering work has been done by Jonkers [18] to define a semantics of pointer manipulating programs that is abstract in the sense of ignoring low-level aspects such as dangling pointers and garbage objects. We explore the principles of such storeless semantics from a logical point of view, first defining a simple logic to completely characterize heap structures up to isomorphism. Second, we extend this language to a full-blown alias logic (AL) that allows to express regular properties of unbounded heap structures. Along the development, we present an operational storeless semantics and give sound and complete total correctness axioms for deterministic programs in the form of Hoare triples, using AL.

Categories and Subject Descriptors

F.3.1 [**Theory of Computation**]: LOGICS AND MEANINGS OF PROGRAMS—*Specifying and Verifying and Reasoning about Programs*

Keywords

heap models, weakest precondition, total correctness

1. INTRODUCTION

This paper provides a formalism for describing properties of linked data structures such as lists, trees and graphs. It also provides an associated program logic for reasoning about programs that destructively update such data structures. The introduced logic, which is called AL for Alias Logic, allows to describe aliases naturally. Moreover, the truth of formulae of this logic is insensitive to garbage collection. Reynolds argues in [21] that program logics such as Hoare calculus based on low level view of storage are incompatible with garbage collection. On the other hand, garbage collection is an essential feature of runtime environments of languages as Lisp, ML and Java. The key issue of our approach is to define a semantic domain for pointer-handling programs that is fully abstract with respect to garbage and renaming of memory locations. This *storeless* domain provides us with the needed means to define a garbage insensitive semantics for Alias Logic.

We consider a set of instructions for altering stacks and heaps and provide three different semantics for these instructions. We also study the relationships between these semantics (See Fig 1). We start from a low level concrete semantics, denoted by $[\![\cdot]\!]_1$ in Fig. 1, where the state of a program is described by a stack and heap. We then show that this semantics does not distinguish between states that are equal up-to the identity of memory locations (addresses). In other words, renaming of locations induces an equivalence relation that is a bisimulation (Diagram (1) in Fig 1). Moreover, we provide a logic characterization of this equivalence relation.

This semantics is, however, sensitive to garbage and dangling pointers from which we would like to abstract. Therefore, we present a store-less semantics $[\![\cdot]\!]_2$ where a heap is modeled as a structure that is a set of regular languages satisfying additional conditions. The idea behind this semantics is that each heap element can be represented by a Rabin-Scott automaton with a single accepting state. Our store-less semantics is akin to Jonker's [18] and Deutsch's [7] semantics. In contrast to them, however, we chose to work explicitly with equivalence classes of an alias relation. This leads to a clean operational semantics based on storeless structures. We also connect with the problem of *symmetry reductions* [6, 11] showing that the structures we introduce are canonical symbolic representations of equivalence classes of states. Next, we show the equivalence between the concrete and the symbolic operational semantics modulo renaming of locations (Diagram (2) in Fig. 1).

The third semantics we introduce is an axiomatic semantics, i.e., a Hoare logic-like proof system. The main feature of this proof system is that it allows to prove properties of programs that are insensitive to garbage. Thus, our proof system is compatible with garbage collection. The problem of designing program logics enjoying this property has been studied by Hoare and Jifeng in [16]. To obtain the desired result they explicitly introduce in the assertion language an operator which models garbage collection, i.e., transforms a state into a garbage free state. A drawback of this approach, as emphasized by Calcagno, O'Hearn and Bornat in [3], is that one has to explicitly carry around a state parameter. These authors propose in [3] an alternative approach which consists in altering the semantics of the assertion language in such a way that the program logic becomes compatible with

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PEPM'03, June 7, 2003, San Diego, California, USA.
Copyright 2003 ACM 1-58113-667-6/03/0006 ...$5.00.

garbage collection. This approach is based on a possible-world interpretation for existential quantification, where the current heap is the world. A similar approach is followed in [5]. Our approach is based on an assertion language that allows to reason explicitly about aliasing. It is called Alias Logic, AL for short. AL allows to describe sets of symbolic structures as introduced in the symbolic garbage insensitive semantics. Interpreting the logic on symbolic structures makes it is insensitive to garbage and allows for designing a weakest precondition calculus that enjoys the same property. We prove soundness and completeness of our program logic (Diagram (3)) and show its applicability on a well known example.

$$\begin{tikzcd}
State \arrow[r, leftrightarrow] \arrow[d, "{[\![\cdot]\!]_1}"] & State/_{\approx} \arrow[r, leftrightarrow] \arrow[d, "{[\![\cdot]\!]_1}"] & \mathcal{S}(\Sigma) \arrow[r, leftrightarrow] \arrow[d, "{[\![\cdot]\!]_2}"] & AL \arrow[d, "\widetilde{pre}"] \\
State \arrow[r, leftrightarrow] & State/_{\approx} \arrow[r, leftrightarrow] & \mathcal{S}(\Sigma) \arrow[r, leftrightarrow] & AL
\end{tikzcd}$$

Diagram (1) Diagram (2) Diagram (3)

Figure 1: Summary of the contributions

1.1 Related Work

Our work tackles the problem of pointer confinement, that is common to the fields of static analysis and software verification. In this context, our definition of the storeless model closely relates to the early work of Jonkers [18] and Deutsch [7]. Hoare and Jifeng provided a trace model to describe shape graphs that matches ours in terms of operational semantics definition. Our approach takes this view further, providing a logical characterization of heap structures and using this logic to also describe program executions, in terms of weakest preconditions.

To describe properties of dynamic program stores, various formalisms have been previously proposed in the literature e.g., Path Matrices [13], ADDS (Abstract Description of Data Structures) [14], L_r [1], BI (Bunched Implications) [17], Separation Logic [21] and PAL (Pointer Assertion Language) [24]. In designing such a language, one has to effectively balance expressibility and decidability, or complexity of the decision procedure. For instance, L_r [1] uses regular expressions to describe reachability between two points in the heap and is shown to be decidable, yet the lack of quantifiers makes one pessimistic about its usefulness as a program calculus. On the other hand, BI [17] and Separation Logic [21] produce remarkably simple preconditions and have quite clean proof-theoretic models [22]. Another feature of these formalisms is that they allow for compositional reasoning [23]. As a downside, the quantifier fragment, essential to express weakest preconditions, is undecidable [4]. Another negative aspect is that these logics are sensitive to garbage, as they inherently rely on a store-based heap model. On one side, our logic abstracts naturally from garbage, not facing the above mentioned problems. Introducing local reasoning in AL seems to be possible too, yet the extension is less intuitive.

From the class of effectively decidable formalisms we mention PAL [24], an extension of second-order monadic logic on trees that allows to describe a restricted (yet interesting) class of graphs, known as "graph types" [19]. Intuitively, graph types are k-successor trees augmented with cross edges, described by regular expressions. Programs that manipulate such graphs are restricted to updating only the underlying tree (backbone). The resulting actions can thus be described in monadic second-order logic, hence the validity of Hoare triples expressed in PAL can be automatically decided [20]. The main difference between PAL and our approach resides in the specification style: PAL specifications are given by structural invariants associated with record types, whereas AL specifications refer, in principle, to the entire heap. Another aspect involves expressibility. PAL seems to be applicable rather to functional programs that manipulate graph types than to unrestricted imperative object-oriented programs. AL was mainly designed for the latter class. We cannot yet compare the formalisms from the decidability point of view, as the satisfiability of an AL formula is still subject of ongoing work.

2. A SIMPLE HEAP LOGIC

This section is dedicated to the definition of a simple logic for performing observations on the heap. This language is propositional logic, where the atomic terms describe pointer aliasing. We name this language heap logic (HL). The abstract syntax of HL is given in Figure 2 (up), and its store-based denotational semantics, in Figure 2 (down). We denote by Σ the set of all pointer variables and by Σ^+ the set of all access paths i.e., non-empty sequences of pointer variables. Since we are interested only in describing shapes, we consider that all variables in a program are pointers. Note that Σ is assumed to contain both global variables and record fields. The distinction between the two classes of variables is made by the context: given a non-empty path $\sigma \in \Sigma^+$, its first element σ_0 always denotes a global variable, whereas the rest of the symbols σ_i $(i > 0)$ are record fields. We can also assume that the programs are well typed and name clashes are resolved at compile time.

$$\begin{array}{rcl}
u, v, x & \in & \Sigma \\
\sigma, \tau, \theta & \in & \Sigma^* \\
 & & \\
f & := & \sigma \diamond \tau \mid f_1 \vee f_2 \mid \neg f
\end{array}$$

$$\begin{array}{rcl}
s & \in & Store \triangleq \Sigma \mapsto Loc_\perp \\
h & \in & Heap \triangleq Loc \mapsto Store_\perp \\
st & \in & State \triangleq Store \times Heap \\
 & & \\
[\![\sigma]\!]_{s,h} & \triangleq & \left\{ \begin{array}{ll} s(v) & \text{if } \sigma \equiv v \\ h([\![\tau]\!]_{s,h}, v) & \sigma \equiv \tau v \\ \perp & \text{otherwise} \end{array} \right. \\
[\![\sigma \diamond \tau]\!]_{st} & \triangleq & [\![\sigma]\!]_{st} = [\![\tau]\!]_{st}
\end{array}$$

Figure 2: The Heap Logic HL

For the semantics, we consider a set *Loc* of memory locations. Note that this is the set of memory addresses, in a "physical" sense. This set is considered infinite but countable and we represent it as $\{l_0, l_1, \dots\}$. As usual, a *store* is a partial mapping between variables and values. Since in our case all variables are pointers, all values are locations. We express the fact that a mapping f is undefined in a point x by $f(x) =\perp$. For a given set A, the notation $A_\perp$ means $A \cup \{\perp\}$; we always assume $\perp \notin A$. A *heap* is a partial mapping between locations and stores. More precisely, given a heap h and a location l, the expression $h(l)$ denotes a store or $\perp$ if $l =\perp$. For a variable x, the notation $h(l, x)$ stands for $h(l)(x)$. We may refer to the stores in the range of a heap as to *objects*. Notice that, since h is a function, one location denotes at most one object, in our setting. We assume that heap functions are strict i.e., $h(\perp) = \perp$. The denotation of the terms and expressions of our language is given with respect to *states*. A state *st* is a pair store-heap (s, h), in which the first component represents the values of *global* variables i.e., variables that are not heap-allocated. Notice that in this setting the denotation of a string in a state $[\![\sigma]\!]_{s,h}$ is a location, or $\perp$ if the access path σ is dangling in the state. Two paths are said to be *aliased* if and only if they lead to the same location. The propositional logic connectives are defined as usual.

To simplify reasoning about states, we consider the function *Reach*, which maps a state *st* and a location l to the set of all access paths reaching l in *st*. Also, the function *Reachable* gives, for a state the set of reachable locations within it.

$$\begin{array}{rcl} \mathit{Reach} & : & \mathit{State} \times \mathit{Loc} \to \mathcal{P}(\Sigma^+) \\ \mathit{Reach}(st, l) & \triangleq & \{\sigma \in \Sigma^+ \mid [\![\sigma]\!]_{st} = l\} \end{array} \quad (1)$$

$$\begin{array}{rcl} \mathit{Reachable} & : & \mathit{State} \to \mathcal{P}(\mathit{Loc}) \\ \mathit{Reachable}(st) & \triangleq & \{l \in \mathit{Loc} \mid \exists \sigma \in \Sigma^+ \; [[\![\sigma]\!]_{st} = l]\} \end{array} \quad (2)$$

Using these functions, we can now define the notions of *total* and *garbage free* states.

Definition 1. *A state $s, h \in \mathit{Store} \times \mathit{Heap}$ is said to be:*

- finite *if and only if the domain of h is finite.*
- total *if and only if $\mathit{Reachable}(s, h) \subseteq \mathit{dom}(h)$, and,*
- garbage free *if and only if $\mathit{dom}(h) \subseteq \mathit{Reachable}(s, h)$.*

Next, we define *garbage collection* on states as the function *gc* which restricts the domain of the heap h in a state (s, h) to the set of the reachable locations. Formally:

$$\begin{array}{rcl} gc & : & \mathit{State} \to \mathit{State} \\ gc(s, h) & \triangleq & s, h\downarrow_{\mathit{Reachable}(s,h)} \end{array} \quad (3)$$

2.1 An Imperative Programming Language

We consider a simple language of atomic statements and let programs be sequences of statements. The abstract syntax of statements is shown in Figure 3. The first statement sets the left-hand side variable to null, which may cause the deletion of non-reachable objects by the garbage collector. The second statement allocates a fresh cell for further uses. The third statement is the assignment operation between variables.

$$\begin{array}{rcll} \mathit{Stmn} & := & \sigma := \mathit{null} & \\ & | & \sigma := \mathit{new} & \\ & | & \sigma := \sigma' & \quad \text{where } \sigma \in \Sigma^+ \end{array}$$

Figure 3: Syntax of Statements

The operational semantics is given in Figure 4. For each statement, we distinguish two cases, depending on the length of the left-hand side path σ. If $|\sigma| = 1$, the statement changes the value of a global variable. Otherwise the statement affects a heap-allocated variable.

$$\frac{}{s, h \overset{v:=null}{\rightsquigarrow} gc(s[v \mapsto \perp], h)} \quad (4)$$

$$\frac{\begin{array}{c} l_{new} \notin \mathit{dom}(h) \\ h_{new} = h[l_{new} \mapsto \lambda x.\perp] \end{array}}{s, h \overset{v:=new}{\rightsquigarrow} gc(s[v \mapsto l_{new}], h_{new})} \quad (5)$$

$$\frac{[\![\theta]\!]_{s,h} = l \qquad l \neq \perp}{s, h \overset{v:=\theta}{\rightsquigarrow} gc(s[v \mapsto l], h)} \quad (6)$$

$$\frac{[\![\tau]\!]_{s,h} = l, \;\; l \neq \perp, \;\; s' = h(l)[v \mapsto \perp]}{s, h \overset{\tau.v:=null}{\rightsquigarrow} gc(s, h[l \mapsto s'])} \quad (7)$$

$$\frac{\begin{array}{ll} [\![\tau]\!]_{s,h} = l, \;\; l \neq \perp & l_{new} \notin \mathit{dom}(h) \\ h_{new} = h[l_{new} \mapsto \lambda x.\perp] & s' = h(l)[v \mapsto l_{new}] \end{array}}{s, h \overset{\tau.v:=new}{\rightsquigarrow} gc(s, h_{new}[l \mapsto s'])} \quad (8)$$

$$\frac{\begin{array}{l} [\![\tau]\!]_{s,h} = l \quad\; l \neq \perp \\ [\![\theta]\!]_{s,h} = l' \quad l' \neq \perp \quad s' = h(l)[v \mapsto l'] \end{array}}{s, h \overset{\tau.v:=\theta}{\rightsquigarrow} gc(s, h[l \mapsto s'])} \quad (9)$$

Figure 4: Operational Semantics

Note that, due to the lack of an explicit delete operation, all states generated by this semantics starting with a total state are total. Moreover, they are also garbage free, since the *gc* function is invoked for each transition. Let $\rightsquigarrow \subseteq \mathit{State} \times \mathit{Stmn} \times \mathit{State}$ be the least transition relation defined by the rules in Figure 4.

3. ON HEAP ISOMORPHISM

The store-based model of computation is redundant. Intuitively, two states that differ only by a renaming of locations are equivalent with respect to formulas written in HL. Moreover, this equivalence is a bisimulation [15] i.e., it has been shown that the direct successors of two equivalent states are also equivalent [12]. We formalize this notion as follows.

Definition 2 (Isomorphism). *Two states are said to be isomorphic, denoted by $s, h \approx s', h'$, if and only if there exists a bijection $\pi : \mathit{Loc} \to \mathit{Loc}$ such that:*

- *$s' = \lambda v.\pi(s(v))$, and*

- $h' = \lambda l.\lambda v.\pi(h(\pi^{-1}(l), v))$.

Intuitively, the values of all variables in the global store and object fields are permuted. The objects in the heap need to be permuted by the inverse permutation, in order to consistently reflect these changes.

Given the semantics in Figure 2, to establish whether two paths in the heap are aliased, we need to compare their denotations. The outcome of this observation is however independent of the actual values of paths, and can be shown to be invariant under isomorphic transformations of states. Hence we can associate each location in the domain of the heap an invariant set of paths, which is the set of all incoming paths.

The following theorem relates states isomorphism and the heap logic HL: two isomorphic states are indistinguishable by any HL formula, and viceversa, two states that are indistinguishable by HL are isomorphic. Due to space limitations, all proofs are deferred to [2].

Theorem 1. *Let $st, st' \in State$ denote two total garbage free states, and f denote any HL formula over the alphabet Σ. Then we have:*

$$st \approx st' \iff \forall f\ [[\![f]\!]_{st} = [\![f]\!]_{st'}]$$

This result shows that, despite its simplicity, the HL language is powerful enough to distinguish non-isomorphic states. We also conjecture a stronger result: for each finite total garbage free state $st \in State$, there exists an HL formula f^{st} that characterizes st up to isomorphism. Investigating this issue into further detail is somehow outside the scope of this paper and is considered for an extended version.

4. A STORELESS MODEL

As mentioned before, the level of detail in the store-based model for the heap is too high. This allows to distinguish between semantically equivalent states. Moreover, this model retains information related to garbage and dangling pointers, from which we would like to abstract. Hence, we introduce a symbolic representation, based on the theory of regular languages. In the new, *storeless* model, a heap is a collection of languages. The idea behind this representation is that each object is the language accepted by the heap graph, viewed as an automaton with that object as a unique final state.

Definition 3 (Storeless structure). *A storeless structure $\Gamma \subseteq \mathcal{P}(\Sigma^+)$ is either the empty set or a set $\{S_1, S_2, \ldots, S_n\}$ satisfying the following conditions, for all $1 \leq i, j \leq n$:*

(C1) non-emptiness: $S_i \neq \emptyset$,

(C2) determinism: $i \neq j \Rightarrow S_i \cap S_j = \emptyset$,

(C3) prefix closure and right regularity: $\forall \sigma \in S_i\ [\forall \tau, \theta \in \Sigma^+[\sigma = \tau\theta \Rightarrow \exists 1 \leq k \leq n\ [\tau \in S_k \wedge S_k\theta \subseteq S_i]]]$.

Let $\mathcal{S}(\Sigma)$ denote the set of all storeless structures over the alphabet Σ. An alternative way of defining a storeless structure is by considering an equivalence relation (alias) on the set of all heap paths. This is the approach taken by Jonkers [18] and Deutsch [8]. By requiring that the equivalence relation be right-regular, they obtain that each language in the heap is recognizable by a finite automaton. Instead, we choose to represent equivalence classes explicitly and impose the right-regularity condition as (C3). Notice that our structures are deterministic (C2) since a path is not allowed to belong to two different sets. Moreover, we exclude empty sets (C1) from our representation; empty sets could serve as an abstract representation of (all) garbage objects which we have chosen to ignore.

By requiring that structures are formed only with non-empty paths ($\Gamma \subseteq \mathcal{P}(\Sigma^+)$) we represent only *rooted* graphs i.e., graphs in which there are no incoming paths towards an initial node. This constraint suits our model of heap well since all paths of length one denote store (local) variables and we allow cycles only in the heap. Given a structure Γ, by Γ_ϵ we denote the set $\Gamma \cup \{\{\epsilon\}\}$.

Let us point now the discussion towards proving the soundness and completeness of the three rules characterizing the storeless semantics. We will do so by relating storeless structures to the previously defined store-based semantics.

Definition 4 (Correspondence). *Let $\Gamma \in \mathcal{S}(\Sigma)$ be a structure and $st = s, h \in State$ be a state. We say that Γ and st correspond, denoted $\Gamma \leftrightharpoons st$, if and only if there exists a bijection $\pi : dom(h) \to \Gamma$ such that:*

1. *for all $u \in \Sigma$, $s(u) = l$ if and only if $u \in \pi(l)$.*
2. *for all $l, l' \in dom(h)$ and $u \in \Sigma$, $h(l, u) = l'$ if and only if $\pi(l)u \subseteq \pi(l')$.*

In principle, we cannot represent a state with garbage or dangling pointers by a corresponding storeless structure, without violating condition (C3). The following lemma shows a method of transforming a store-based state into a storeless structure. Moreover, it states that there is only one way to do so. Our construction is in fact the equivalent of the left quotienting in automata theory [10].

Lemma 1. *Let $st = s, h \in State$ be a total garbage free state. Then the set $\{Reach(st, l) \mid l \in dom(h)\}$ is a storeless structure. Moreover, if $\Gamma \in \mathcal{S}(\Sigma)$ is a storeless structure such that $st \leftrightharpoons \Gamma$, then $\Gamma = \{Reach(st, l) \mid l \in dom(h)\}$.*

This result implies that a storeless structure can be used as a *canonical* symbolic representation of isomorphic states. The following theorem is the first important result of this section. It postulates the correctness of conditions (C1), (C2) and (C3) by finding, for each store-based state a corresponding storeless structure and viceversa, for each storeless structure a corresponding state.

Theorem 2. *For each total garbage free state $st \in State$ there exists a unique structure $\Gamma \in \mathcal{S}(\Sigma)$ such that $st \leftrightharpoons \Gamma$. Dually, for each structure $\Gamma \in \mathcal{S}(\Sigma)$ there exists a garbage free total state $st \in State$ such that $st \leftrightharpoons \Gamma$.*

This theorem expresses the fact that our definition of storeless structures has the same expressive power as the right-regular equivalences used by Jonkers [18] and Deutsch [8] and the trace model described by Hoare and Jifeng [16]: we are now capable to describe rooted directed labeled graphs. However, the explicit use of regular languages, enables us to give an operational semantics on storeless structures that is easier to understand and implement using finite automata. This is the discussion point of the next subsection. Next, in Section 5, we develop an alias logic that uses regular expressions too, and for which deciding whether a given structure is a model of a given formula boils down to checking emptiness of a regular language.

4.1 Storeless Operational Semantics

Having defined storeless structures as a symbolic representation for states, we can now define program actions as operations on regular languages. But first let us introduce Eilenberg's "division" operator for languages [10]. Given two sets $S, T \subseteq \Sigma^*$ we denote by $S^{-1}T$ the set $\{\sigma \in \Sigma^+ \mid S\sigma \cap T \neq \emptyset\}$. Note that, if $S, T \in \Gamma$ for some $\Gamma \in \mathcal{S}(\Sigma)$, $S^{-1}T = \{\sigma \in \Sigma^+ \mid S\sigma \subseteq T\}$[1]. By $\mathcal{P}^2(\Sigma^*)$ we denote the set $\mathcal{P}(\mathcal{P}(\Sigma^*))$.

In order to simplify the presentation, we define first three primitive transformations, and later, present the full semantics using compositions of the primitive actions. Informally, *rem* (10) describes the effect of removing an arc v from a graph node represented by the language S. Notice that eliminating a single arc removes a possibly infinite number of paths from the structure, potentially introducing garbage objects. These objects are automatically represented by an empty set, which is finally eliminated from the structure. Next, *new* (11) is used to model object creation. From an origin node S we create a new node represented by Sv and add it to the structure. The most complex operation is *add* (12) which adds a (possibly new) arc between two nodes S and T. This complexity is an inherent consequence of the fact that cycles might be introduced. Nevertheless, all the transformations occurring in *add* are easily implemented via automata. Interestingly, our semantics for *add* matches exactly Hoare and Jifeng's semantics for pointer swing [16]. The rest of this section is concerned with proving the correctness of the storeless operational semantics.

$$
\begin{aligned}
\mathit{rem} \quad &: \quad \mathcal{P}(\Sigma^*) \times \Sigma \to \mathcal{P}^2(\Sigma^*) \to \mathcal{P}^2(\Sigma^*) \\
\mathit{rem}(S, v) \quad &\triangleq \quad \lambda\Gamma.\{X \setminus Sv\Sigma^* \mid X \in \Gamma\} \setminus \{\emptyset\} \qquad (10)
\end{aligned}
$$

$$
\begin{aligned}
\mathit{new} \quad &: \quad \mathcal{P}(\Sigma^*) \times \Sigma \to \mathcal{P}^2(\Sigma^*) \to \mathcal{P}^2(\Sigma^*) \\
\mathit{new}(S, v) \quad &\triangleq \quad \lambda\Gamma.\Gamma \cup \{Sv\} \qquad (11)
\end{aligned}
$$

$$
\begin{aligned}
\mathit{add} \quad &: \quad \mathcal{P}(\Sigma^*) \times \Sigma \times \mathcal{P}(\Sigma^*) \to \mathcal{P}^2(\Sigma^*) \to \mathcal{P}^2(\Sigma^*) \\
\mathit{add}(S, v, T) \quad &\triangleq \quad \lambda\Gamma.\{\chi^{S,v,T}(X) \mid X \in \Gamma\} \text{ where,} \qquad (12) \\
\chi^{S,v,T}(X) \quad &\triangleq \quad X \cup Sv((T^{-1}S)v)^*(T^{-1}X)
\end{aligned}
$$

Notice first that the primitive operations can be applied to any languages $S, T \subseteq \Sigma^*$, any symbol $v \in \Sigma$ and any set of languages $\Gamma \in \mathcal{P}(\Sigma^*)$. The following three lemmas are then the first steps in our correctness proof. Assuming that we start with a storeless structure $\Gamma \in \mathcal{S}(\Sigma)$, and two languages in $S, T \in \Gamma_\epsilon$, we postulate necessary and sufficient conditions for the result of the three operations to be a valid storeless structure.

Intuitively, removing a v arc from a node S will further remove all paths prefixed by an element of Sv. In particular, there are no side conditions for *rem*.

[1]This is true in general when S and T represent equivalence classes of a congruence relation on paths.

Lemma 2. *Let $\Gamma \in \mathcal{S}(\Sigma)$ be a storeless structure and $S \in \Gamma_\epsilon$ be a set. Then, for all symbols $v \in \Sigma$ we have $rem(S, v, \Gamma) \in \mathcal{S}(\Sigma)$.*

Intuitively, creating a new object pointed to by an arc v that originates in S, is equivalent to adding the $\{Sv\}$ set to the structure. The *new* operation yields a correct result if and only if there are no common paths between the new node and some existing node or the new node already existed in the structure, in which case nothing is changed. Notice that, if this condition is violated the result will necessarily be non-deterministic, hence violate the (C2) condition.

Lemma 3. *Let $\Gamma \in \mathcal{S}(\Sigma)$ be a storeless structure and $S \in \Gamma_\epsilon$ be a set. Then, for all symbols $v \in \Sigma$ we have $new(S, v, \Gamma) \in \mathcal{S}(\Sigma)$ if and only if for all $T \in \Gamma$, either $Sv \cap T = \emptyset$ or $T = Sv$.*

The *add* operation is the most complex one, because it might introduce cycles into the structure. We capture cycles by the Kleene closure in the formula (12). Adding an arc v from S to T will induce a cycle generated by $(T^{-1}S)v$ i.e., all paths originating in T that end in S concatenated with v. Note that no cycle is introduced when $T^{-1}S = \emptyset$ since, by convention $\emptyset^* = \{\epsilon\}$. The last term from the expression of $\chi^{S,v,T}$ denotes the influence of a cycle on an arbitrary set X. Note that X is not affected by *add* if there are no paths from T to X.

Before proving soundness with respect to the store-based model, we give a fixpoint formulation for the $\chi^{S,v,T}$ function, used in the definition of *add* (12); while the ready-made formula (12) is easy to implement, the fixpoint formulation will be of more use in reasoning about the correctness of *add*. To improve readability, we skip the superscripts of χ. Given three languages S, T and X, we define the following function:

$$\xi_X(z) \triangleq X \cup Sv(T^{-1}z) \qquad (13)$$

Now let us show that $\chi(X) = \text{fix } z.\xi_X$. Let $Y \triangleq T^{-1}\text{fix } z.\xi_X$. We have:

$$
\begin{aligned}
Y &= T^{-1}(X \cup Sv(T^{-1}\text{fix } z.\xi_X)) \\
&= T^{-1}X \cup T^{-1}(Sv(T^{-1}\text{fix } z.\xi_X)) \\
&= T^{-1}X \cup (T^{-1}S)vY
\end{aligned}
$$

We have used that $T^{-1}(Sv) = (T^{-1}S)v$, which is easily checked. Also $\epsilon \notin (T^{-1}S)v$, and, by Arden's lemma, we obtain that $Y = ((T^{-1}S)v)^*(T^{-1}X)$ is the unique solution to the above equation. Since $\chi(X) = X \cup SvY$, we have the result. It is easy to check that the ξ_X function is affine i.e., for any $\mathcal{Y} \in \mathcal{P}^2(\Sigma^*)$ we have $\xi_X(\bigcup \mathcal{Y}) = \bigcup \xi_X(\mathcal{Y})$. Hence $\chi(X) = \bigcup_{i \geq 0} \xi_X^i(\emptyset)$.

The *add* operation yields a correct result if and only if the newly added arc from S to T does not already exist between S and a node different than T. If this condition would be violated the resulting structure would be non-deterministic.

Lemma 4. *Let $\Gamma \in \mathcal{S}(\Sigma)$ be a storeless structure and $S \in \Gamma_\epsilon$, $T \in \Gamma$ be two sets. Then, for all symbols $v \in \Sigma$ we have $add(S, v, T, \Gamma) \in \mathcal{S}(\Sigma)$ if and only if for all $T' \in \Gamma$, either $Sv \cap T' = \emptyset$ or $T' = T$.*

Figure 5 presents the operational semantics of the three statements defined by the syntax in Figure 3. As in the definition of the store-based semantics, we treat separately the case where the left-hand side of the assignment is a local variable v or a path $\tau.v$ of length two or more. In the first case, the first argument of *rem*, *new* and *add* is ϵ. Otherwise, we need to identify a node S in the source structure to which the v variable belongs. In order to keep the semantics small, we do not treat null pointer dereferencing errors.

$$\frac{}{\Gamma \overset{v:=null}{\hookrightarrow} rem(\epsilon, v, \Gamma)} \qquad (14)$$

$$\frac{}{\Gamma \overset{v:=new}{\hookrightarrow} (new(\epsilon, v) \circ rem(\epsilon, v))(\Gamma)} \qquad (15)$$

$$\frac{\exists S\ [S \in \Gamma \wedge \theta \in S]}{\Gamma \overset{v:=\theta}{\hookrightarrow} (add(\epsilon, v, S) \circ rem(\epsilon, v))(\Gamma)} \qquad (16)$$

$$\frac{\exists S\ [S \in \Gamma \wedge \tau \in S]}{\Gamma \overset{\tau.v:=null}{\hookrightarrow} rem(S, v, \Gamma)} \qquad (17)$$

$$\frac{\exists S\ [S \in \Gamma \wedge \tau \in S]}{\Gamma \overset{\tau.v:=new}{\hookrightarrow} (new(S, v) \circ rem(S, v))(\Gamma)} \qquad (18)$$

$$\frac{\exists S, T\ [S, T \in \Gamma \wedge \tau \in S \wedge \theta \in T]}{\Gamma \overset{\tau.v:=\theta}{\hookrightarrow} (add(S, v, T) \circ rem(S, v))(\Gamma)} \qquad (19)$$

Figure 5: Storeless Operational Semantics

To use the primitive operations previously defined, we need to make sure that the side conditions stated in Lemma 3 and 4 are actually met. This is accomplished using *rem* before *new* or *add* to first clear the "inconsistent" paths from the structure. In the case of pointer assignment (rules (16) and (19)), removing first some paths from the structure leads to a well known problem: if the left-hand side path is a prefix of the right-hand side and there are no other incoming paths to the right-hand side node, then this node will be eliminated before the assignment takes place. A solution proposed in the literature [18, 16] uses *fresh* paths that are explicitly added to the right-hand side node and removed after the assignment. To simplify the semantics, we precompile our program introducing a fresh temporary variable whenever $\tau.v$ is a prefix of θ i.e., transforming the statement $\tau.v := \theta$ into $v_{fresh} := \theta$; $\tau.v := v_{fresh}$. Notice that the prefix condition is just a syntax check easily performed during program parsing. Let $\hookrightarrow \subseteq \mathcal{S}(\Sigma) \times Stmn \times \mathcal{S}(\Sigma)$ be the relation defined by the rules in Figure 5.

THEOREM 3. *Let $m \in Stmn$ be a statement, $st \in State$ be a total garbage free state and $\Gamma \in \mathcal{S}(\Sigma)$ be a structure such that $st \leftrightharpoons \Gamma$. If $st \overset{m}{\rightsquigarrow} st'$ for some $st' \in State$ then there exists $\Gamma' \in \mathcal{S}(\Sigma)$ such that $\Gamma \overset{m}{\hookrightarrow} \Gamma'$ and $st' \leftrightharpoons \Gamma'$. Dually, if $\Gamma \overset{m}{\hookrightarrow} \Gamma'$ for some $\Gamma' \in \mathcal{S}(\Sigma)$, then there exists $st' \in State$ such that $st \overset{m}{\rightsquigarrow} st'$ and $st' \leftrightharpoons \Gamma'$.*

PROOF. Let $st = s, h$ be a total garbage free state and Γ a storeless structure such that $st \leftrightharpoons \Gamma$. By Lemma 1, since st is total and garbage free, we obtain $\Gamma = \{Reach(st, l) \mid l \in dom(h)\}$. We denote by $st' = s', h'$ the m-successor of st if one exists i.e., $st \overset{m}{\rightsquigarrow} st'$, and by Γ' the m-successor of Γ if one exists i.e., $\Gamma \overset{m}{\hookrightarrow} \Gamma'$. It is sufficient to prove that st' exists if and only if Γ' exists, and when both exist, $\Gamma' = \{Reach(st', l) \mid l \in dom(h')\}$. By Lemma 1 this entails $st' \leftrightharpoons \Gamma'$. In order to check the equivalence of the preconditions, observe that

$$[\![\tau]\!]_{st} = l \iff \exists S \in \Gamma\ [\tau \in S]$$

is immediate from the fact that $\Gamma = \{Reach(st, l) \mid l \in dom(h)\}$. To show equivalence of actions, we distinguish the following cases, depending on the structure of m (we present the proofs only for the more interesting cases, when the left-hand side of the assignment is $\tau_0.v$):

- m is $\tau_0.v := null$. By rule (7) we have $dom(h) = dom(h')$. For each path $\sigma \in \Sigma^+$ and each location $l \in dom(h)$, it can be proven by induction on the length of σ that $[\![\sigma]\!]_{st'} = l$ iff:
$$[\![\sigma]\!]_{st} = l \wedge \neg(\exists \tau, \theta\ [\sigma = \tau v \theta \wedge [\![\tau]\!]_{st} = [\![\tau_0]\!]_{st}])$$
In other words, all paths from st are preserved in st', except if they pass through the arc v originating in $[\![\tau]\!]_{st}$. With the notation $S \triangleq Reach(st, [\![\tau_0]\!]_{st})$ we obtain that $Reach(st', l) = Reach(st, l) \setminus Sv\Sigma^*$. Hence $\Gamma' = rem(S, v, \Gamma) = \{Reach(st', l) \mid l \in dom(h')\}$.

- m is $\tau_0 v := new$. Let l_{new} be the new location defined by rule (8). Hence $dom(h') = dom(h) \cup \{l_{new}\}$. Moreover, all paths in st are strictly preserved in st'. New paths leading to l_{new} are introduced and they must end through the arc v originating in $[\![\tau_0]\!]_{st}$. Formally, for each path $\sigma \in \Sigma^+$, we can show by induction on the length σ that $\sigma \in Reach(st', l')$ iff:
$$\begin{gathered}(l \neq l_{new}\ \wedge\ \sigma \in Reach(st, l)) \vee \\ (l = l_{new}\ \wedge\ \exists \tau[\sigma = \tau v \wedge [\![\tau]\!]_{st} = [\![\tau_0]\!]_{st}])\end{gathered}$$
Hence, the set $Reach(st', l) = Reach(st, l)$ for any $l \neq l_{new}$ and respectively $Reach(st', l_{new}) = Sv$ where $S \triangleq Reach(st, [\![\tau_0]\!]_{st})$. Hence $\Gamma' = new(S, v, \Gamma) = \{Reach(st', l) \mid l \in dom(h')\}$.

- m is $\tau_0.v := \theta_0$. By rule (9) we have $dom(h) = dom(h')$. For each path $\sigma \in \Sigma^+$ and each location $l \in dom(h)$, it can be proven by induction on σ that $[\![\sigma]\!]_{st'} = l$ iff:
$$[\![\sigma]\!]_{st} = l \vee \exists \tau, \theta\ [\sigma = \tau v \theta \wedge [\![\tau]\!]_{st} = [\![\tau_0]\!]_{st} \wedge [\![\theta_0 \theta]\!]_{st'} = l]$$
In other words, σ existed already in st or it was added by the transition. Hence the set $Reach(st', l)$ is a solution of the following equation:
$$\begin{aligned} X \quad = \quad & Reach(st, l) \cup \\ & Reach(st, [\![\tau_0]\!]_{st}) v (Reach(st, [\![\theta_0]\!]_{st})^{-1} X) \end{aligned}$$
With the notation $X_0 \triangleq Reach(st, l)$, $S \triangleq Reach(st, [\![\tau_0]\!]_{st})$ and $T \triangleq Reach(st, [\![\theta_0]\!]_{st})$ we obtain that $X = \text{fix}\ \xi_{X_0}$, where ξ_X is defined as (13). As previously discussed, this equation has a unique solution which is given by $\chi(X) = X \cup Sv((T^{-1}S)v)^*(T^{-1}X)$. Hence $\Gamma' = add(S, v, T, \Gamma) = \{Reach(st', l) \mid l \in dom(h')\}$.

□

This proves soundness of the storeless semantics with respect to the classical store-based model.

5. ALIAS LOGIC

In this section we describe the full-blown alias logic AL which embeds our initial heap logic HL and which is next developed into a program logic. Being able to reason only about aliases is not enough for a precondition calculus. Therefore AL has a modality operator which allows to specify into which node in the graph paths may flow. Since nodes are given as recognizable languages, the decision of modalities boils down to deciding language emptiness of product automata.

Let Var denote a set of free variables, $Reg(Var, \Sigma)$ denote the set of all regular expressions over Σ containing variables from Var, and $Term(Var, \Sigma)$ denote the set of all terms built out of regular expressions, equality operator and a modality operator together with the classic connectives of first-order logic. Figure 6 gives the syntax (up) and the semantics (down) of AL.

$$\begin{array}{rcl} u & \in & \Sigma \\ X & \in & Var \\ \rho & \in & Reg(Var, \Sigma) \\ \varphi & \in & Term(Var, \Sigma) \\ \\ \rho & ::= & u \mid \Sigma \mid X \mid \rho_1 \cdot \rho_2 \mid \rho^* \\ & & \mid \rho_1 \cup \rho_2 \mid \rho_1 \cap \rho_2 \mid \bar{\rho} \mid \rho_1^{-1}\rho_2 \\ \varphi & ::= & \rho_1 = \rho_2 \mid \langle \rho_1 \rangle \rho_2 \\ & & \mid \varphi_1 \vee \varphi_2 \mid \neg\varphi \mid \exists X\ [\varphi] \end{array}$$

$$\begin{array}{rcl} \Gamma & \in & \mathcal{S}(\Sigma) \\ free & : & Reg(Var, \Sigma) \cup Term(Var, \Sigma) \\ & & \to \mathcal{P}(Var) \\ \nu & : & Var \to \Sigma^* \\ \\ [\![\rho]\!]_\nu & \triangleq & \rho[\vec{X}/\nu(\vec{X})] \text{ where } \vec{X} = free(\rho) \\ [\![\rho_1 = \rho_2]\!]_\nu & \triangleq & [\![\rho_1]\!]_\nu = [\![\rho_2]\!]_\nu \\ [\![\langle\rho_1\rangle\rho_2]\!]_{\Gamma,\nu} & \triangleq & [\![\rho_1]\!]_\nu \in \Gamma \wedge [\![\rho_1]\!]_\nu \cap [\![\rho_2]\!]_\nu \neq \emptyset \\ [\![\exists X\ [\varphi]]\!]_\nu & \triangleq & \exists \rho \in \mathcal{P}(\Sigma^*)\ [\![\varphi]\!]_{[X \to \rho]\nu} \end{array}$$

Figure 6: The Alias Logic AL

In the following, we will refer to the $\langle\rho_1\rangle\rho_2$ terms as to *modalities*. To ease notation, we introduce some syntactic shortcuts: $\rho_1 \setminus \rho_2 \triangleq \rho_1 \cap \bar{\rho_2}$, $\emptyset \triangleq \rho \setminus \rho$ (for some $\rho \subseteq \Sigma^*$) and $\rho_1 \subseteq \rho_2 \triangleq \rho_1 \cap \bar{\rho_2} = \emptyset$. Using disjunction, negation and existential quantification, we can define the rest of logical connectives as $true \triangleq \varphi \vee \neg\varphi$, $false \triangleq \neg true$, $\varphi_1 \wedge \varphi_2 \triangleq \neg(\neg\varphi_1 \vee \varphi_2)$, $\varphi_1 \Rightarrow \varphi_2 \triangleq \neg\varphi_1 \vee \varphi_2$, and $\forall X\ [\varphi] \triangleq \neg\exists X\ [\neg\varphi]$.

Given a term, the *free* function returns the set of free variables occurring within its regular expressions. This function is usually defined by induction on the structure of terms i.e., $free(\bullet\varphi) \triangleq free(\varphi)$ for each unary term and $free(\varphi_1 \bullet \varphi_2) \triangleq free(\varphi_1) \cup free(\varphi_2)$ for each binary term. Existential quantification eliminates free variables i.e., $free(\exists X\ [\varphi]) \triangleq free(\varphi) \setminus \{X\}$. A structure Γ is said to be a *model* for a term φ, denoted $\Gamma \models \varphi$ if and only if $[\![\varphi]\!]_{\Gamma,\lambda X.\epsilon}$ is true. We define *pure assertions* to be formulas not containing modalities. Notice that the semantics of a pure assertion is given independently of a heap structure i.e., if φ is pure then either $\forall\ \Gamma \in \mathcal{S}(\Sigma)\ [\Gamma \models \varphi]$ or $\forall\ \Gamma \in \mathcal{S}(\Sigma)\ [\Gamma \not\models \varphi]$.

Note that, despite the fact that quantifiers range over sets of paths, AL remains first-order logic. Indeed, we do not allow quantification over the elements of a set $X \in \mathcal{P}(\Sigma^*)$.

We shall now introduce further handy notation: $\rho_1 \diamond \rho_2 \triangleq \exists X\ [\langle X\rangle\rho_1 \wedge \langle X\rangle\rho_2]$ to express the may-aliasing between two regular paths ρ_1 and ρ_2, and $in(X) \triangleq \langle X\rangle\Sigma^*$ to express the presence of node X in the storeless structure. The fact that a node Y is reachable from another node X can be defined as $reach(X, Y) \triangleq in(X) \wedge in(Y) \wedge X\Sigma^* \cap Y \neq \emptyset$. The fact that X and Y belong to a cycle is expressed as $cycle(X, Y) \triangleq reach(X, Y) \wedge reach(Y, X)$.

It is worthwhile pointing out that, by defining the $\diamond$ predicate, we have embedded the HL logic into AL. Hence AL can be used to describe sets of store-based states as well as sets of storeless configurations, according to Theorem 3. Next, we present examples of AL formulas that describe common place heap structures. The next subsection will discuss the use of AL to define the semantics of programs, in Hoare style.

Examples To show the use of AL as a language for describing the shape of pointer structures, we consider the following predicates:

$$\begin{array}{rcl} nclist(h, n) & \triangleq & \exists X\ [\langle X\rangle h] \wedge \forall X, Y \\ & & [X \cup Y \subseteq h.n^* \wedge X \diamond Y \Rightarrow X = Y] \\ nshared(h_1, h_2, n) & \triangleq & \neg(h_1.n^* \diamond h_2.n^*) \\ tree(root) & \triangleq & \exists X\ [\langle X\rangle root] \wedge \forall X, Y \\ & & [root.X \diamond root.Y \Rightarrow X = Y] \\ dag(root) & \triangleq & \exists X\ [\langle X\rangle root] \wedge \forall Y, Z \\ & & [reach(X, Y) \wedge reach(X, Z) \\ & & \Rightarrow \neg cycle(Y, Z)] \end{array}$$

The *nclist* predicate is true in all states in which there exists a possibly empty non-circular list pointed to by the variable h. The non-circularity requirement is captured by the fact that if two paths are aliased, then they must be the equal. The *nshared* predicate is true when there is no sharing between a list starting with h_1 and a list starting with h_2, if they both use the same selector n. A *tree* structure is described by the lack of sharing within all the nodes reachable from the node pointed to by the *root* variable. To describe a *dag* we only require that there are no cycles between the nodes reachable from the top node.

AL has obvious limitations due to the expressive power of regular expressions. For instance, describing a balanced tree could only be done imposing the restriction that any two paths leading towards leaves have the same length. To cope with these problems, extensions of AL towards context free and tree languages are considered as future work.

5.1 Axiomatic Semantics

Having introduced a logic to represent sets of storeless configurations, we tackle now the problem of using this logic to compute weakest preconditions. We define hereby sound and complete inference rules to characterize the execution of the three statements we have considered throughout the paper. In this setting we deal with total correctness i.e., our assertions distinguish statements that go "wrong" from the ones that execute correctly.

Definition 5 (Weakest Precondition). *Given an AL term φ and a statement $m \in Stmn$, define $wp(m, \varphi) \subseteq \mathcal{S}(\Sigma)$ to be the largest set such that if $\Gamma \in wp(m, \varphi)$ then there exists $\Gamma' \in \mathcal{S}(\Sigma)$ such that $\Gamma \overset{m}{\hookrightarrow} \Gamma'$ and $\Gamma' \models \varphi$.*

We recall a number of classical results [9] on weakest preconditions seen as predicate transformers i.e., the set $wp(\omega, \varphi)$ being characterized by a first-order predicate $\widetilde{pre}(\omega, \varphi)$. For any transition relation over a sequence of statements ω, $\widetilde{pre}$ distributes over conjunction and universal quantification i.e., $\widetilde{pre}(\omega, \varphi_1 \wedge \varphi_2) = \widetilde{pre}(\omega, \varphi_1) \wedge \widetilde{pre}(\omega, \varphi_2)$ and $\widetilde{pre}(\omega, \forall X\ [\varphi]) = \forall X\ [\widetilde{pre}(\omega, \varphi)]$. For total transition relations we have $\widetilde{pre}(\omega, \varphi) \Rightarrow \neg\widetilde{pre}(\omega, \neg\varphi)$. If, moreover, the transition relation is total and deterministic, we have that $\widetilde{pre}$ is its own dual i.e., $\widetilde{pre}(\omega, \varphi) \Leftrightarrow \neg\widetilde{pre}(\omega, \neg\varphi)$. In the latter case $\widetilde{pre}$ distributes over disjunction and existential quantification too.

These properties of $\widetilde{pre}$ for total deterministic programs allow us to define general inference rules for the precondition inductively on the structure of the postcondition. Therefore, we can first give sound and complete characterizations of *wp* for the primitive storeless operations *rem* (10) and *add* (12) in cases where the postconditions are modalities only. Then we can generalize to arbitrary postconditions using the distributivity of $\widetilde{pre}$ operators over first-order connectives in case of deterministic programs. Next, we will generalize the axioms to describe $\widetilde{pre}$ for all statements in Figure 3. In conclusion, we discuss the treatment of non-deterministic programs in AL.

Remove The following rule defines the weakest precondition of a modality formula with respect to a removal operation.

$$\{\exists X\ [X \setminus Sv\Sigma^* = T \wedge \langle X\rangle(\sigma \setminus Sv\Sigma^*)]\}\ \mathbf{rem(S,v)}\ \{\langle T\rangle\sigma\} \tag{20}$$

We show now that the remove rule is sound and complete, by proving the following lemma. Although *rem* is a primitive transformation and not a statement, we still denote by $wp(rem(S, v), \varphi)$ the largest set of structures $\Gamma \in \mathcal{S}(\Sigma)$ which, under the transformation $rem(S, v)$, lead to a structure satisfying φ. According to Lemma 2, we need to assume that $S \in \Gamma$, as a side condition, otherwise the structure obtained from Γ by applying $rem(S, v)$ might not be consistent with Definition 3.

Lemma 5. *Given $\Gamma, \Gamma' \in \mathcal{S}(\Sigma)$ two structures such that $\Gamma' \models \langle T\rangle\sigma$, and $S \in \Gamma$ a set, then $\Gamma \in wp(rem(S, v), \langle T\rangle\sigma)$ if and only if $\Gamma \models \exists X\ [X \setminus Sv\Sigma^* = T \wedge \langle X\rangle\sigma \setminus Sv\Sigma^*]$.*

Proof. The proof is based on the following equivalence:

$$(\sigma \setminus Sv\Sigma^*) \cap X = \sigma \cap (X \setminus Sv\Sigma^*) \tag{21}$$

"$\Rightarrow$" Let $\Gamma \in wp(rem(S, v), \langle T\rangle\sigma)$ be a storeless structure. Then, by Lemma 2 and since $S \in \Gamma$, $\Gamma' = rem(S, v, \Gamma)$ is also a structure and $\Gamma' \models \langle T\rangle\sigma$. Then $T \in \Gamma'$ and $X \setminus Sv\Sigma^* = T$ for some $X \in \Gamma$. By (21), we have $\sigma \cap T = (\sigma \setminus Sv\Sigma^*) \cap X$. Hence $\Gamma \models \exists X\ [X \setminus Sv\Sigma^* = T \wedge \langle X\rangle\sigma \setminus Sv\Sigma^*]$. "$\Leftarrow$" Let $\Gamma \models \exists X\ [X \setminus Sv\Sigma^* = T \wedge \langle X\rangle\sigma \setminus Sv\Sigma^*]$ and $\Gamma' = rem(S, v, \Gamma)$. Then $X \in \Gamma$ and $T \in \Gamma'$ by rule (10). By (21) we have that $\Gamma' \models \langle T\rangle\sigma$. Hence $\Gamma \in wp(rem(S, v), \langle T\rangle\sigma)$. $\Box$

New This rule defines the weakest precondition of a modality with respect to the new object creation operation.

$$\{(T = Sv \wedge \sigma \cap Sv \neq \emptyset) \vee \langle T\rangle\sigma\}\ \mathbf{new(S,v)}\ \{\langle T\rangle\sigma\} \tag{22}$$

It can be shown that the rule above is sound and complete with respect to the storeless operational semantics. Using the same abuse of notation, we denote by $wp(new(S, v), \varphi)$ the largest set of structures $\Gamma \in \mathcal{S}(\Sigma)$ which, under the transformation $new(S, v)$, lead to a structure satisfying φ. We need to also assume the necessary and sufficient condition from Lemma 3 in order to ensure that the result of $new(S, v)$ is a consistent storeless structure.

Lemma 6. *Given $\Gamma, \Gamma' \in \mathcal{S}(\Sigma)$ two structures such that $\Gamma' \models \langle T\rangle\sigma$, and $S \in \Gamma$ a set such that for all $Y \in \Gamma$, either $Sv \cap Y = \emptyset$ or $Y = Sv$, then $\Gamma \in wp(new(S, v), \langle T\rangle\sigma)$ if and only if $\Gamma \models (T = Sv \wedge \sigma \cap Sv \neq \emptyset) \vee \langle T\rangle\sigma$.*

Proof. "$\Rightarrow$" Let $\Gamma \in wp(new(S, v), \langle T\rangle\sigma)$ be a storeless structure such that $S \in \Gamma$ and for all $Y \in \Gamma$, either $Sv \cap Y = \emptyset$ or $Y = Sv$. If we denote $\Gamma' = new(S, v, \Gamma)$, by Lemma 3, we have $\Gamma' \in \mathcal{S}(\Sigma)$. Also $\Gamma' \models \langle T\rangle\sigma$, if and only if $T \in \Gamma' = \Gamma \cup \{Sv\}$ (by rule (11)) and $T \cap \sigma \neq \emptyset$. Then either $T \in \Gamma$ or $T = Sv$. Hence $\Gamma \models (T = Sv \wedge \sigma \cap Sv \neq \emptyset) \vee \langle T\rangle\sigma$. "$\Leftarrow$" Assume that $\Gamma \models (T = Sv \wedge \sigma \cap Sv \neq \emptyset) \vee \langle T\rangle\sigma$ and let $\Gamma' = rem(S, v, \Gamma)$. Then either $T = Sv \wedge \sigma \cap Sv \neq \emptyset$ is true or $\Gamma \models \langle T\rangle\sigma$. Both cases imply $\Gamma' \models \langle T\rangle\sigma$, by rule (11). $\Box$

Add The last rule defines the weakest precondition for modalities under the edge *add* operation. The complexity of the precondition formula occurs as an inherent consequence of *add*'s rather complex storeless operational semantics (12).

$$\{\exists X\ [\chi(X) = U \wedge \bigvee_{i=1,2} \psi_i(X)]\}\ \mathbf{add(S,v,T)}\ \{\langle U\rangle\sigma\} \tag{23}$$

where

$$\psi_1(X) \triangleq Sv((T^{-1}S)v)^*(T^{-1}X) \cap \sigma = \emptyset \wedge \langle X\rangle\sigma$$
$$\psi_2(X) \triangleq Sv((T^{-1}S)v)^*(T^{-1}X) \cap \sigma \neq \emptyset \wedge \langle X\rangle\Sigma^*$$

The soundness and completeness proof is done in a similar way, with $wp(add(S, v, T), \varphi)$ denoting the weakest precondition with respect to *add* and the side condition from Lemma 4, added to ensure consistency of the result.

Lemma 7. *Given $\Gamma, \Gamma' \in \mathcal{S}(\Sigma)$ two structures such that $\Gamma' \models \langle U\rangle\sigma$, and $S, T \in \Gamma$ two sets such that, for all $Y \in \Gamma$, either $Sv \cap Y = \emptyset$ or $Y = T$, then $\Gamma \in wp(add(S, v, T), \langle U\rangle\sigma)$ if and only if $\Gamma \models \exists X\ [\chi(X) = U \wedge \bigvee_{i=1,2} \psi_i(X)]$.*

Proof. The proof is based on the following equivalences, which result immediately from rule (12). For any $X \in \Sigma^*$ such that $\chi(X) = U$:

$$U \cap \sigma = (Sv((T^{-1}S)v)^*(T^{-1}X) \cap \sigma) \cup (X \cap \sigma)$$
$$U \cap \sigma \neq \emptyset \iff \psi_1(X) \vee \psi_2(X) \tag{24}$$

"$\Rightarrow$" Let $\Gamma \in wp(add(S,v,T), \langle U\rangle\sigma)$ be a storeless structure such that for all $Y \in \Gamma$, either $Sv \cap Y = \emptyset$ or $Y = T$. If we denote $\Gamma' = add(S,v,T,\Gamma)$, by Lemma 4, we have $\Gamma' \in \mathcal{S}(\Sigma)$ and $\Gamma' \models \langle U\rangle\sigma$. So $U \in \Gamma'$ and, by rule (12), $\chi(X) = U$ for some $X \in \Gamma$. Also, by (24) we have $\psi_1(X) \vee \psi_2(X)$. "$\Leftarrow$" Assume that $\Gamma \models \exists X\ [\chi(X) = U \wedge \bigvee_{i=1,2} \psi_i(X)]$ and let $\Gamma' = add(S,v,T,\Gamma)$. If $\psi_i(X)$, $i = 1,2$ holds on Γ we have $X \in \Gamma$. By rule (12) we have then $U \in \Gamma'$. Applying (24) we obtain $\Gamma' \models \langle U\rangle\sigma$. $\square$

It is to be noticed that, in the above claims, we have implicitly used the fact that all primitive operations on the storeless heap are total functions i.e., Γ' always exists. This observation leads to the fact that the transition relation $\hookrightarrow$ defined by the rules in Figure 5 is both total and deterministic. According to the previous discussion, the weakest precondition predicate transformer $\widetilde{pre}$ distributes over all first order logical connectives. Under this assumption, we can express the precondition of an arbitrary AL formula φ recursively on the structure of φ. Let *op* be a primitive operation i.e., one of *rem*, *new* and *add* provided with some sound parameters S, v and T, then we denote by $\widetilde{pre}(op,\varphi)$ the formula obtained by recursively applying rules (20), (22) and (23) to φ. Notice that, if φ is a pure assertion, we have $\widetilde{pre}(op,\varphi) = \varphi$. With these considerations, Figure 7 shows the weakest preconditions for the three types of statements considered in this paper. In order to deal with total correctness, for statements that use dereferencing, we must add conditions to match the preconditions of the operational semantic rules in Figure 5. Differently stated, this ensures that all transitions can actually execute. For a sequence of statements $\omega \in Stmn^*$ let, $\widetilde{pre}(\omega,\varphi)$ be the precondition formula defined by the rules in Figure 7 using the classical composition rule $\{\widetilde{pre}(m, \widetilde{pre}(n,\varphi))\}$ **m**; **n** $\{\varphi\}$.

$$\begin{gathered}\{\widetilde{pre}(rem(\epsilon,v),\varphi)\}\\ v := null\\ \{\varphi\}\end{gathered}$$

$$\begin{gathered}\{\widetilde{pre}(rem(\epsilon,v),\widetilde{pre}(new(\epsilon,v),\varphi))\}\\ v := new\\ \{\varphi\}\end{gathered}$$

$$\begin{gathered}\{\exists S\ [\langle S\rangle\theta \wedge \widetilde{pre}(rem(\epsilon,v),\widetilde{pre}(add(\epsilon,v,S),\varphi))]\}\\ v := \theta\\ \{\varphi\}\end{gathered}$$

$$\begin{gathered}\{\exists S\ [\langle S\rangle\tau \wedge \widetilde{pre}(rem(S,v),\varphi)]\}\\ \tau.v := null\\ \{\varphi\}\end{gathered}$$

$$\begin{gathered}\{\exists S\ [\langle S\rangle\tau \wedge \widetilde{pre}(rem(S,v),\widetilde{pre}(new(S,v),\varphi))]\}\\ \tau.v := new\\ \{\varphi\}\end{gathered}$$

$$\begin{gathered}\{\exists S,T\ [\langle S\rangle\tau \wedge \langle T\rangle\theta \wedge\\ \widetilde{pre}(rem(S,v),\widetilde{pre}(add(S,v,T),\varphi))]\}\\ \tau.v := \theta\\ \{\varphi\}\end{gathered}$$

Figure 7: Weakest Preconditions for Statements

THEOREM 4. *Given $\Gamma,\Gamma' \in \mathcal{S}(\Sigma)$ two structures, $\omega \in Stmn^*$ a statement and φ an AL formula such that $\Gamma' \models \varphi$, then $\Gamma \in wp(\omega,\varphi)$ if and only if $\Gamma \models \widetilde{pre}(\omega,\varphi)$.*

PROOF. By induction on the structure of φ. $\square$

In conclusion, we briefly discuss the use of AL to describe the semantics of non-deterministic programs. In practice, non-determinism can be the result of parallel composition i.e., one can imagine a parallel version of the language in Figure 3, or abstraction i.e., non-deterministic choices can be introduced by loss of precision. In terms of weakest preconditions, non-determinism means that the implication $\widetilde{pre}(\omega,\varphi) \Leftarrow \neg\widetilde{pre}(\omega,\neg\varphi)$ does not hold any longer. Consequently we also lose the fact that $\widetilde{pre}(\omega,\varphi_1 \vee \varphi_2) \Rightarrow \widetilde{pre}(\omega,\varphi_1) \vee \widetilde{pre}(\omega,\varphi_2)$ and the same for existential quantifier. However, $\widetilde{pre}(\omega,\varphi_1 \vee \varphi_2) \Leftarrow \widetilde{pre}(\omega,\varphi_1) \vee \widetilde{pre}(\omega,\varphi_2)$ still holds, and similar for the existential quantifier. This results in a loss of completeness of weakest preconditions. Notice that, if we still replace $\widetilde{pre}(\omega,\varphi_1 \vee \varphi_2)$ by $\widetilde{pre}(\omega,\varphi_1) \vee \widetilde{pre}(\omega,\varphi_2)$ we obtain a stronger precondition i.e., a sound but incomplete rule. In practice this might be useful still, since any result that we can infer is correct.

However, the reasoning we applied to disjunction and existential quantification cannot be applied to negation, since replacing $\widetilde{pre}(\omega,\neg\varphi)$ by $\neg\widetilde{pre}(\omega,\varphi)$ results in weakening the precondition. Instead, we write the postcondition in positive normal form (with only atomic terms in the scope of a negation) and give a set of sound axioms for the negated modalities. This construction is further developed in the Appendix (A). It is easy to verify that using positive normal forms with sound axioms for negation, ensures soundness of the precondition axioms.

5.2 An Example

We assessed our calculus on a classical example in the literature [21]: the in-place list reversal program from Figure 8. The goal is to prove that $nclist(i,n) \wedge nclist(j,n) \wedge nshared(i,j,n)$ is an invariant of the while loop. Since we deal with total correctness, we shall assume a side condition of the form $\exists I\ [\langle I\rangle i]$ throughout the computation. In other words, we assume that the i-list has at least one element, which allows us to iterate at least once.

```
j := null;
while i ≠ null do
        k := i.n;
        i.n := j;
        j := i;
        i := k;
od
```

Figure 8: List Reversal Program

Below we give the bottom-up derivation for the loop body. Each step of the derivation is the result of a non-trivial series of implications. For presentation purposes we have skipped the simplifications applied. The interested reader is referred to the Appendix (B) for an example. We use here the consequence rule i.e., if $P \Rightarrow Q$ and $\{Q\}$ **C** $\{R\}$ then $\{P\}$ **C** $\{R\}$.

$$\{nclist(i,n) \wedge nclist(j,n) \wedge nshared(i,j,n)\}$$
$$\Rightarrow$$
$$\{nclist(i,n) \wedge nclist(j,n) \wedge nshared(i,j,n) \wedge$$
$$\forall I, J\ [\langle I \rangle i \wedge \langle J \rangle j \Rightarrow J^{-1} I \cap n^* = \emptyset]\}$$
$$k := i.n$$
$$\{nclist(k,n) \wedge nclist(j,n) \wedge nshared(k,j,n) \wedge$$
$$\forall I, J\ [\langle I \rangle i \wedge \langle J \rangle j \Rightarrow J^{-1} I \cap n^* = \emptyset]\}$$
$$i.n := j$$
$$\{nclist(k,n) \wedge nclist(i,n) \wedge nshared(k,i,n)\}$$
$$j := i$$
$$\{nclist(k,n) \wedge nclist(j,n) \wedge nshared(k,j,n)\}$$
$$i := k$$
$$\{nclist(i,n) \wedge nclist(j,n) \wedge nshared(i,j,n)\}$$

The first two steps can be, in our particular case, resumed by substitution (see Appendix B for more detail). In the third step we obtain an extra precondition saying that there should be no n-paths between the node containing j and the node containing i, or else the assignment $i.n := j$ would generate a cycle, invalidating $nclist(i,n)$. This precondition is however implied by $nshared(i,j,n)$ and disappears when we apply the consequence rule after the last step.

6. CONCLUSIONS AND FUTURE WORK

This paper presents an abstract semantic model, based on regular languages that is transparent to the existence of garbage. On top of this model, an assertion language and an associated program logic is developed, which allows to reason about linked data structures. The presented program logic is shown to be sound and complete for program instructions that destructively update such data structures.

The expressive power as well as the decidability of the assertion language AL or of some relevant fragments need to be studied in the future. This is important for our approach to be effectively useful. For instance, finding a non-trivial subset of AL which is both decidable and closed under the weakest precondition transformer is an important tool for the development of an abstract interpretation of pointer handling programs using AL assertions as abstract domain, in the spirit of shape analysis [25].

7. REFERENCES

[1] M. Benedikt, T. Reps and M. Sagiv: A Decidable Logic for Describing Linked Data Structures. In Proc. European Symposium On Programming (1999).

[2] Marius Bozga, Radu Iosif and Yassine Lakhnech: Storeless Semantics and Alias Logic. Technical report, http://www-verimag.imag.fr/~bozga/ssal.ps

[3] C. Calcagno and P.W. O'Hearn: On Garbage and Program Logic. Foundations of Software Science and Computation Structures, Lecture Notes in Computer Science, Volume 2030 (2001) pp 137-151

[4] C. Calcagno, H. Yang and P.W. O'Hearn: Computability and Complexity Results for a Spatial Assertion Language for Data Structures. In FST TCS 2001: Foundations of Software Technology and Theoretical Computer Science, Lecture Notes in Computer Science, Volume 2245 (2001), pp 108-119

[5] Frank S. de Boer. Reasoning about dynamically evolving process structures. Ph.D. Thesis, Vrije Universiteit te Amsterdam (University of Amsterdam), 1991.

[6] Edmund M. Clarke, Somesh Jha, Reinhard Enders and Thomas Filkorn: Exploiting Symmetry In Temporal Logic Model Checking. Formal Methods in System Design, Vol.9, No. 1/2 (1996) 77–104

[7] Alain Deutsch. A storeless model of aliasing and its abstractions using finite representations of right-regular equivalence relations. In Proceedings of the IEEE 1992 Conference on Computer Languages (April 1992) pp. 2-13

[8] Alain Deutsch. Interprocedural May-Alias Analysis for Pointers: Beyond k-limiting. In Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI), Orlando, Florida, June 20-24, 1994. SIGPLAN Notices 29(6) (June 1994) pp. 230-241

[9] Edsger W. Dijkstra and Carel S. Scholten. Predicate Calculus and Program Semantics. Springer- Verlag, New York 1990.

[10] Samuel Eilenberg. Automata, Languages, and Machines. Academic Press (1976)

[11] E. Emerson and A. P. Sistla: Symmetry and Model Checking. Formal Methods in System Design, Vol.9, No. 1/2 (1996) 105–131

[12] Radu Iosif: Symmetry Reductions for Model Checking of Concurrent Dynamic Software. Submitted, http://www-verimag.imag.fr/~iosif/papers.html

[13] R. Ghiya and L. Hendren: Is it a Tree, a DAG or a Cyclic Graph? A Shape Analysis for Heap-Directed Pointers in C. In Proc. 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, (1996).

[14] L. Hendren, J. Hummel and A. Nicolau: Abstractions for Recursive Pointer Data Structures: Improving the Analysis and Transformation of Imperative Programs. In Proc. ACM SIGPLAN Conference on Programming Languages Design and Implementation, (1992).

[15] M. Hennessy and R. Milner: Algebraic Laws for Nondeterminism and Concurrency. Journal of the ACM Vol. 32 (1985) pp. 137-161

[16] C.A.R Hoare and He Jifeng: A Trace Model for Pointers and Objects. In Proc. ECOOP'99, Lecture Notes in Computer Science, Vol. 1628, pp. 1-18 (1999)

[17] Samin Ishtiaq and Peter O'Hearn: BI as an Assertion Language for Mutable Data Structures. Proc. of 28th ACM-SIGPLAN Symposium on Principles of Programming Languages (2001)

[18] H. B. M. Jonkers. Abstract Storage Structures. Algorithmic Languages, North-Holland (1981)

[19] N. Klarlund and M. I. Schwartzbach: In Proc. 20th Annual Symposium on Principles of Programming Languages (1993) pp. 196-205

[20] N. Klarlund and M. I. Schwartzbach: Graphs and Decidable Transductions Based on Edge Constraints, In Proc. 19th Colloquium on Trees and Algebra in Programming, Lecture Notes in Computer Science, Volume 787 (1994) pp. 187-201

[21] John C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. Proc 17th IEEE Symposium on Logic in Computer Science (2002)

[22] P.W. O'Hearn and D.J. Pym: The Logic of Bunched Implications. Bulletin of Symbolic Logic, 5(2) (1999) pp. 215-244

[23] P.W. O'Hearn, J.C. Reynolds and H. Yang: Local reasoning about programs that alter data structures. Computer Science Logic, Lecture Notes in Computer Science, Volume 2142 (2001) pp 1-19

[24] A. Moeller and M. I. Schwartzbach: The Pointer Assertion Logic Engine. In Proc. ACM SIGPLAN Conference on Programming Languages Design and Implementation, (2001).

[25] M. Sagiv, M., T. Reps and R. Wilhelm. Parametric Shape Analysis via 3-Valued Logic. Conference Record of the Twenty-Third ACM Symposium on Principles of Programming Languages, (San Antonio, TX, Jan. 20-22, 1999), ACM, New York, NY, 1999.

APPENDIX

A. RULES FOR NEGATION

The following axioms express sound preconditions for the negated modalities. This is useful for developing a sound (but not complete) semantics of non-deterministic programs. Notice first that $\neg\langle U\rangle\sigma = \neg in(U) \vee \sigma \cap V = \emptyset$. Since $\sigma \cap U = \emptyset$ is a pure assertion, it is sufficient to define preconditions only for $\neg in(U)$. Note that the axioms below can be easily shown to be also complete. The overall lack of completeness of the non-deterministic semantics occurs when inferring the precondition of a disjunctive formula i.e., $\widetilde{pre}(\omega, \varphi_1 \vee \varphi_2) \Leftarrow \widetilde{pre}(\omega, \varphi_1) \vee \widetilde{pre}(\omega, \varphi_2)$. Disjunctive formulas are obtained, for instance, by applications of the *add* weakest precondition rule (23).

Negate Remove Intuitively, a set U will not belong to a structure after a remove has been performed iff by performing the remove operation from an existing set we will not obtain U.

$$\{\forall X\ [in(X) \Rightarrow X \setminus Sv\Sigma^* \neq U]\}\ \mathbf{rem(S,v)}\ \{\neg in(U)\}$$

Negate New A set U will not belong to a structure after a new operation has been performed iff it does not belong to the structure before and it is not equal to the set that will be added by *new*.

$$\{\neg in(U) \wedge U \neq Sv\}\ \mathbf{new(S,v)}\ \{\neg in(U)\}$$

Negate Add A set U will not belong to a structure after an add operation has been performed iff the pre-image of its transformation does not belong to the original structure.

$$\{\forall X\ [in(X) \Rightarrow \chi(X) \neq U]\}\ \mathbf{add(S,v,T)}\ \{\neg in(U)\}$$

By adding the $\sigma \cap U = \emptyset$ disjunct to the above preconditions, we will obtain the weakest precondition of $\neg\langle U\rangle\sigma$.

B. DERIVATION EXAMPLE

The following gives an example of reasoning in Alias Logic. We prove that $\{\exists X[\langle X\rangle k]\}\ \mathbf{i := k}\ \{\exists X[\langle X\rangle i]\}$, using that assignment is a composition of *rem* and *add*. By the rules in Figure 7 we have:

$$\begin{gathered}\{\exists K\ [\langle K\rangle k \wedge \widetilde{pre}(rem(\epsilon, i), \widetilde{pre}(add(\epsilon, i, K), \langle X\rangle i))]\}\\ i := k\\ \{\langle X\rangle i\}\end{gathered}$$

Next, we calculate: $\widetilde{pre}(add(\epsilon, i, K), \langle X\rangle i) =$

$$\begin{gathered}\exists X'\ [\chi(X') = X \wedge i(K^{-1}X') \cap i = \emptyset \wedge \langle X'\rangle i] \vee\\ \exists X'\ [\chi(X') = X \wedge i(K^{-1}X') \cap i \neq \emptyset \wedge \langle X'\rangle \Sigma^*]\end{gathered}$$

Applying the rule for *rem* to the first disjunct we obtain, for some X' and X'':

$$\begin{gathered}X'' \setminus i\Sigma^* = X' \wedge \chi(X') = X \wedge\\ i(K^{-1}X') \cap i = \emptyset \wedge \langle X''\rangle(i \setminus i\Sigma^*)\end{gathered}$$

Notice that the last conjunct is false since $i \setminus i\Sigma^* = \emptyset$ and $X'' \cap \emptyset = \emptyset$. Hence we are left with the following term, obtained by an application of the *rem* rule to the second disjunct above:

$$\begin{gathered}X'' \setminus i\Sigma^* = X' \wedge \chi(X') = X \wedge\\ i(K^{-1}X') \cap i \neq \emptyset \wedge \langle X''\rangle\overline{i\Sigma^*})\end{gathered}$$

The third conjunct yields $\epsilon \in K^{-1}X'$, therefore $K \cap X' \neq \emptyset$. If $\langle K\rangle k$ and $\langle X'\rangle\Sigma^*$, by (C2) we have that $K = X'$. Hence $k \in X'$. If $X'' \setminus i\Sigma^* = X'$ then $k \in X''$. Hence $\langle X''\rangle k$ for some X''.

Using that $\{\exists X[\langle X\rangle k]\}\ \mathbf{i := k}\ \{\exists X[\langle X\rangle i]\}$ in a compositional fashion, we can easily obtain the first step of the derivation in Section 5.2.