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 heap 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 [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 \([\cdot, \cdot]_2\) 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 sgraeomety 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 Borel 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
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 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.

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 σ ∈ Σ*, its first element σ₀ always denotes a global variable, whereas the rest of the symbols σᵢ (i > 0) are record fields. We can also assume that the programs are well typed and name clashes are resolved at compile time.

\[ u, v, x \in \Sigma \]
\[ \sigma, \tau, \theta \in \Sigma^* \]
\[ f ::= \sigma \bowtie \tau \mid f_1 \lor f_2 \mid \neg f \]

\[ s \in \text{Store} \triangleq \Sigma \rightarrow \text{Loc}_\perp \]
\[ h \in \text{Heap} \triangleq \text{Loc} \rightarrow \text{Store}_\perp \]
\[ st \in \text{State} \triangleq \text{Store} \times \text{Heap} \]

\[ [\sigma; \tau]_{s;h} \triangleq \begin{cases} s(v) & \text{if } \sigma \equiv v \\ h([\sigma]_{s;h}, v) & \sigma \equiv \tau v \\ \bot & \text{otherwise} \end{cases} \]

\[ [\sigma \bowtie \tau]_{st} \triangleq [\sigma]_{st} \]

\begin{figure}[h]
\centering
\caption{The Heap Logic HL}
\end{figure}
For the semantics, we consider a set \( \text{Loc} \) 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 \( \{l_0, l_1, \ldots \} \). 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 \( f(x) = \bot \). For a given set \( A \), the notation \( A \cup \{ \bot \} \) means \( A \cup \{ \bot \} \); we always assume \( \bot \not\in A \). 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 \( \sigma \) or \( \bot \) if \( l = \bot \). 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., \( h(\bot) = \bot \). The denotation of the terms and expressions of our language is given with respect to states. A state \( s \) 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 \( \bot \) if the access path \( \sigma \) 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 \( \text{Reach} \), which maps a state \( s \) and a location \( l \) to the set of all access paths reaching \( l \) in \( s \). Also, the function \( \text{Reachable} \) gives, for a state the set of reachable locations within it.

\[
\begin{align*}
\text{Reach} : \quad & \text{State} \times \text{Loc} \to \mathcal{P}(\Sigma^+) \\
\text{Reach}(s, l) & \triangleq \{ \sigma \in \Sigma^+ \mid [\sigma]_{s,h} = l \} \\
\text{Reachable} : \quad & \text{State} \to \mathcal{P}(\text{Loc}) \\
\text{Reachable}(s) & \triangleq \{ l \in \text{Loc} \mid \exists \sigma \in \Sigma^+ \left[ [\sigma]_{s,h} = l \right] \}
\end{align*}
\]

(1)

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

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

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

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:

\[
\begin{align*}
\text{gc} : \quad & \text{State} \to \text{State} \\
\text{gc}(s, h) & \triangleq s, h \upharpoonright \text{Reachable}(s, h)
\end{align*}
\]

(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.

\[
\begin{align*}
\text{Stmt} & \quad \triangleright \quad \sigma := \text{null} \\
\text{Stmt} & \quad \triangleright \quad \sigma := \text{new} \\
\text{Stmt} & \quad \triangleright \quad \sigma := \sigma' \quad \text{where } \sigma' \in \Sigma^+
\end{align*}
\]

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 \( \sigma \). If \([\sigma] = 1\), the statement changes the value of a global variable. Otherwise the statement affects a heap-allocated variable.

\[
\begin{align*}
\sigma, h & \xrightarrow{\text{new }} \sigma := \text{null} \quad \text{gc}(s[v \mapsto \bot], h) \\
\sigma, h & \xrightarrow{\text{new }} \sigma \quad \text{gc}(s[v \mapsto \sigma'], h) \\
\sigma, h & \xrightarrow{\text{new }} \sigma' \quad \text{gc}(s[v \mapsto \sigma'], h) \\
\sigma, h & \xrightarrow{\text{new }} \sigma' \quad \text{gc}(s[v \mapsto \sigma'], h)
\end{align*}
\]

(4)

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 \( \sim \subseteq \text{State} \times \text{Stmt} \times \text{State} \) 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 \( \pi : \text{Loc} \to \text{Loc} \) such that:

- \( s' = \lambda v. \pi(s(v)) \), and
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 $\Gamma \in \mathcal{P}(\Sigma^*)$ is either the empty set or a set $\{S_1, S_2, \ldots, S_n\}$ satisfying the following conditions, for all $1 \leq i, j \leq n$:

1. **(C1) non-emptiness**: $S_i \neq \emptyset$, 
2. **(C2) determinism**: $i \neq j \Rightarrow S_i \cap S_j = \emptyset$, 
3. **(C3) prefix closure and right regularity**: $\forall \sigma \in S_i \forall \tau, \theta \in \Sigma^* \left[ \sigma = \tau \theta \Rightarrow \exists 1 \leq k \leq n \left[ \tau \in S_k \land S_k \theta \subseteq S_i \right] \right]$.

Let $S(\Sigma)$ denote the set of all storeless structures over the alphabet $\Sigma$. 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 directed labeled 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 $\Gamma$, by $\Gamma'$ 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 $\Gamma \in S(\Sigma)$ be a structure and $st = s, h \in \text{State}$ be a state. We say that $\Gamma$ and $st$ correspond, denoted $\Gamma \models st$, if and only if there exists a bijection $\pi: \text{dom}(h) \rightarrow \Gamma$ such that:

1. for all $u \in \Sigma$, $s(u) = 1$ if and only if $u \in \pi(\ell)$.
2. for all $l, l' \in \text{dom}(h)$ 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 $st = s, h \in \text{State}$ be a total garbage free state. Then the set $\{\text{Reach}(st, l) \mid l \in \text{dom}(h)\}$ is a storeless structure. Moreover, if $\Gamma \in S(\Sigma)$ is a storeless structure such that $st \models \Gamma$, then $\Gamma = \{\text{Reach}(st, l) \mid l \in \text{dom}(h)\}$.

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 $st \in \text{State}$ there exists a unique structure $\Gamma \in S(\Sigma)$ such that $st \models \Gamma$. Dually, for each structure $\Gamma \in S(\Sigma)$ there exists a garbage free total state $st \in \text{State}$ such that $st \models \Gamma$.

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
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 $\Gamma \in S(\Sigma)$, $S^{-1}T = \{ \sigma \in \Sigma^+ \mid S \sigma \subseteq T \}^1$. By $P^2(\Sigma^*)$ we denote the set $P(P(\Sigma^*))$.

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, $\text{rem}$ (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.

\[
\begin{align*}
\text{rem} : \quad P(\Sigma^*) \times \Sigma & \rightarrow P^2(\Sigma^*) \rightarrow P^2(\Sigma^*) \\
\text{rem}(S,v) & \triangleq \lambda \Gamma, \{ X \setminus \Sigma \sigma \mid \sigma \in \Gamma \setminus \{ \emptyset \} \} \tag{10}
\end{align*}
\]

\[
\begin{align*}
\text{new} : \quad P(\Sigma^*) \times \Sigma & \rightarrow P^2(\Sigma^*) \rightarrow P^2(\Sigma^*) \\
\text{new}(S,v) & \triangleq \lambda \Gamma, \{ Sv \} \tag{11}
\end{align*}
\]

\[
\begin{align*}
\add : \quad P(\Sigma^*) \times \Sigma \times P(\Sigma^*) & \rightarrow P^2(\Sigma^*) \rightarrow P^2(\Sigma^*) \\
\add(S,v,T) & \triangleq \lambda \Gamma, (\text{add}_{S,v,T}(X) \mid X \in \Gamma) \tag{12}
\end{align*}
\]

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 $\Gamma \in P(\Sigma^*)$. The following three lemmas are then the first steps in our correctness proof. Assuming that we start with a storeless structure $\Gamma \in S(\Sigma)$, and two languages in $S,T \in \Gamma_{v}$ 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 $\text{rem}$.

---

\[ \text{Lemma 2. Let } \Gamma \in S(\Sigma) \text{ be a storeless structure and } S \in \Gamma_v \text{ be a set. Then, for all symbols } v \in \Sigma \text{ we have } \text{rem}(S,v,T) \in S(\Sigma). \]

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.

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

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 $\emptyset = \{ \emptyset \}$. The last term from the expression of $\text{add}_{S,v,T}$ 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 $\text{add}_{S,v,T}$ (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 $\chi$.

Given three languages $S, T$ and $X$, we define the following function:

\[
\chi_{X}(z) \triangleq X \cup Sv(T^{-1}z) \tag{13}
\]

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

\[
\begin{align*}
Y &= T^{-1}(X \cup Sv(T^{-1}\text{fix } z. \chi_z)) \\
&= T^{-1}X \cup T^{-1}(Sv(T^{-1}\text{fix } z. \chi_z)) \\
&= T^{-1}X \cup (T^{-1}Sv)Y
\end{align*}
\]

We have used that $T^{-1}(Sv) = (T^{-1}S)v$, which is easily checked. Also $\emptyset \in (T^{-1}S)v$, 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 $\chi_z$ function is affine i.e., for any $Y \in P^2(\Sigma^*)$, we have $\chi_z(\bigcup Y) = \bigcup \chi_z(Y)$. Hence $\chi(X) = \bigcup_{z \in \Sigma} \chi_z(z)$. 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.

\[ \text{Lemma 4. Let } \Gamma \in S(\Sigma) \text{ be a storeless structure and } S \in \Gamma_v, T \in \Gamma \text{ be two sets. Then, for all symbols } v \in \Sigma \text{ we have } \text{add}(S,v,T,\Gamma) \in S(\Sigma) \text{ if and only if for all } T' \in \Gamma, \text{ either } S \cap T' = \emptyset \text{ or } T' = T. \]

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 \( \text{rem}, \text{new}, \text{add} \) is \( e \). 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.

\[
\begin{align*}
\Gamma \vdash e : \text{null} & \quad \text{(14)} \\
\Gamma \vdash e : \text{new}(e, v, \Gamma) & \quad \text{(15)} \\
\exists S \ [S \in \Gamma \land \tau \in S] & \quad \text{(16)} \\
\Gamma \vdash \text{null} \quad \text{new}(S, v, \Gamma) & \quad \text{(17)} \\
\exists S \ [S \in \Gamma \land \tau \in S] & \quad \text{(18)} \\
\Gamma \vdash \text{null} \quad \text{new}(S, v, \Gamma) & \quad \text{(19)}
\end{align*}
\]

**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 \( \text{rem} \) before \( \text{new} \) or \( \text{add} \) 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 added to the right-hand side node and removed after the assignment. To simplify the semantics, we pre-compose our program introducing a fresh temporary variable whenever \( \tau.v \) is a prefix of \( \theta \) i.e., transforming the statement \( \tau.v := \theta \) into \( v.fresh := \theta; \tau.v := v.fresh \). Notice that the prefix condition is just a check easily performed during program parsing. Let \( \leftrightarrow \in S(\Sigma) \times S(\Sigma) \times S(\Sigma) \) be the relation defined by the rules in Figure 5.

**Theorem 3.** Let \( m \in \text{Stmn} \) be a statement, \( st \in \text{State} \) be a total garbage free state and \( S \in \Sigma \) be a structure such that \( st \rightarrow \Gamma \). If \( st \vdash st' \) for some \( st' \in \text{State} \) then there exists \( S' \in S(\Sigma) \) such that \( S \rightarrow S' \Gamma \). Dually, if \( S \vdash S' \) for some \( S' \in S(\Sigma) \), then there exists \( st' \in \text{State} \) such that \( st \vdash st' \) and \( S' \rightarrow \Gamma' \).

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

\[
[\sigma]_st = l \iff S \in \Gamma [\tau \in S]
\]

is immediate from the fact that \( \Gamma = \{ \text{Reach}(st, l) \mid l \in \text{dom}(h) \} \). 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 \( \tau_0.v) \):

- \( m = \tau_0.v := \text{null} \). By rule (7) we have \( \text{dom}(h) = \text{dom}(h') \). For each path \( \sigma \in \Sigma^+ \) and each location \( l \in \text{dom}(h) \), it can be proven by induction on the length of \( \sigma \) that \( [\sigma]_st = l \iff \exists \tau \theta \ [\sigma = \tau_0\theta \land [\tau]_st = [\tau_0]_st] \).

In other words, all paths from \( st \) are preserved in \( st' \), except if they pass through the arc \( \nu \) originating in \( [\tau]_st \). With the notation \( S \triangleq \text{Reach}(st, [\tau]_st) \) we obtain that \( \text{Reach}(st', l) = \text{Reach}(st, l) \setminus \text{Se}^\nu S \). Hence \( \Gamma' = \text{rem}(S, v, \Gamma) \rightarrow \{ \text{Reach}(st', l) \mid l \in \text{dom}(h') \} \).

- \( m = \tau_0.v := \text{new} \). Let \( \text{lenw} \) be the new location defined by rule (8). Hence \( \text{dom}(h') = \text{dom}(h) \cup \text{lenw} \). Moreover, all paths in \( st \) are strictly preserved in \( st' \). New paths leading to \( \text{lenw} \) are introduced and they must end through the arc \( \nu \) originating in \( [\tau]_st \). Formally, for each path \( \sigma \in \Sigma^+ \), we can show by induction on the length of \( \sigma \) that \( \exists \tau \theta \ [\sigma \rightarrow \sigma' \land [\tau]_st = [\tau_0]_st] \).

Hence, the set \( \text{Reach}(st', l) = \text{Reach}(st, l) \setminus \text{Se}^\nu S \) for any \( l \neq \text{lenw} \) and respectively \( \text{Reach}(st', \text{lenw}) = \text{Se}^\nu S \) where \( S \triangleq \text{Reach}(st, [\tau]_st) \). Hence \( \Gamma' = \text{new}(S, v, \Gamma) \rightarrow \{ \text{Reach}(st', l) \mid l \in \text{dom}(h') \} \).

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

$$u \in \Sigma$$
$$X \in \text{Var}$$
$$\rho \in \text{Reg}(\text{Var}; \Sigma)$$
$$\varphi \in \text{Term}(\text{Var}; \Sigma)$$

$$\rho ::= u \mid | \rho_1 \cup \rho_2 \mid \rho_1 \cap \rho_2 \mid \rho_1 \setminus \rho_2$$

$$\varphi ::= \rho \mid \rho_1 \cup \rho_2 \mid \rho \cap \rho_2$$

Let $\Gamma \in S(\Sigma)$

$$\text{free} : \text{Reg}(\text{Var}; \Sigma) \cup \text{Term}(\text{Var}; \Sigma)$$

$$\nu : \text{Var} \rightarrow \Sigma^*$$

$$[\rho]_{\nu} \triangleq \rho[\check{X} / \nu(\check{X})] \text{ where } X \equiv \text{free}(\rho)$$

$$[\rho_1 = \rho_2]_{\nu} \triangleq [\rho_1]_{\nu} = [\rho_2]_{\nu}$$

$$[\rho_1 \cup \rho_2]_{\nu} \triangleq [\rho_1]_{\nu} \cup [\rho_2]_{\nu} \neq \emptyset$$

$$[\exists X \varphi]_{\nu} \triangleq \exists \rho \in \text{Reg}(\Sigma^*) \ [\varphi[\check{X} \setminus \nu]]$$

Figure 6: The Alias Logic AL

In the following, we will refer to the $[\rho_1]_{\nu}$ terms as to modalities. To ease notation, we introduce some syntactic shortcuts: $\rho_1 \setminus \rho_2 \triangleq \rho_1 \cap \rho_2 \emptyset \triangleq \rho \setminus \rho$ (for some $\rho \subseteq \Sigma^*$) and $\rho_1 \subseteq \rho_2 \triangleq \rho_1 \cap \rho_2 = \emptyset$. Using disjunction, negation and existential quantification, we can define the rest of logical connectives as true $\triangleq \varphi \lor \neg \varphi$, false $\triangleq \neg true$, $\varphi_1 \land \varphi_2 \triangleq \neg (\neg \varphi_1 \lor \neg \varphi_2)$, $\varphi_1 \rightarrow \varphi_2 \triangleq \neg \varphi_1 \lor \varphi_2$, 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., $\text{free}(\cdot \varphi) \triangleq \text{free}(\varphi)$ for each unary term and $\text{free}(\varphi_1 \cdot \varphi_2) \triangleq \text{free}(\varphi_1) \cup \text{free}(\varphi_2)$ for each binary term. Existential quantification eliminates free variables i.e., $\text{free}(\exists X \varphi) \triangleq \text{free} \setminus \{X\} \setminus \varphi \setminus \{X\}$. A structure $\Gamma$ is said to be a model for a term $\varphi$, denoted $\Gamma \models \varphi$, if and only if $\forall \varphi \in _{\Gamma \models \varphi}$.

We will now introduce further handy notions: $\rho_1 \setminus \rho_2 \triangleq \exists X \{X[\rho_1 \cap \check{X}] \check{X} \setminus \rho_2\}$ to express the may-aliasing between two regular paths $\rho_1$ and $\rho_2$ and $\text{in}(X) \triangleq \text{free}(X)$ 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 $\text{reach}(X, Y) \triangleq \text{in}(X) \land \text{in}(Y) \land X \check{X} \cap Y \neq \emptyset$. The fact that $X$ and $Y$ belong to a cycle is expressed as $\text{cycle}(X, Y, Y)$ $\triangleq \text{reach}(X, Y) \land \text{reach}(Y, X)$.

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:

$$\text{nclist}(h, n) \triangleq \exists X \{X[h] \land \forall Y \leq X \diamond Y \Rightarrow X = Y\}$$

$$\text{nsshared}(h_1, h_2, n) \triangleq \neg (h_1[\text{n} \cap h_2[\text{n}])$$

$$\text{tree}(\text{root}) \triangleq \exists X \{X[\text{root}] \land \forall Y \leq X \diamond Y \Rightarrow X = Y\}$$

$$\text{dag}(\text{root}) \triangleq \exists X \{X[\text{root}] \land \forall Y \leq Z \Rightarrow [\text{reach}(X, Y) \land \text{reach}(X, Z) \Rightarrow \neg \text{cycle}(Y, Z)]\}$$

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 nsshared 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 distinction statements that go “wrong” from the ones that execute correctly.

**Definition 5 (Weakest Precondition).** Given an AL term \( \varphi \) and a statement \( m \in Strm \), define \( wp(m, \varphi) \subseteq S(\Sigma) \) to be the largest set such that if \( \Gamma \vdash wp(m, \varphi) \) then there exists \( \Gamma' \in S(\Sigma) \) such that \( \Gamma \vdash \Gamma' \) 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 \( \overline{wp}(\omega, \varphi) \). For any transition relation over a sequence of statements \( \omega \) \( \overline{wp} \) distributes over conjunction and universal quantification i.e., \( \overline{wp}(\omega_1 \land \varphi_1, \varphi_2) = \overline{wp}(\omega_1, \varphi_1) \land \overline{wp}(\omega_2, \varphi_2) \) and \( \overline{wp}(\omega, \forall X \varphi) = \forall X \overline{wp}(\omega, \varphi) \). For total transition relations we have \( \overline{wp}(\omega, \varphi) \Rightarrow \overline{wp}(\omega, \overline{\varphi}) \). If, moreover, the transition relation is total and deterministic, we have that \( \overline{wp} \) is its own dual i.e., \( \overline{wp}(\omega, \varphi) \Leftrightarrow \overline{wp}(\omega, \overline{\varphi}) \). In the latter case \( \overline{wp} \) distributes over disjunction and existential quantification too.

These properties of \( \overline{wp} \) 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 \( \overline{wp} \) operators over first-order connectives in case of deterministic programs. Next, we will generalize the axioms to describe \( \overline{wp} \) 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.

\[
\exists X \left[ X \setminus S \not\in \Sigma \right] \Rightarrow \left( T \land \left\langle X \left( \sigma \setminus S \not\in \Sigma \right) \right) \right] \Rightarrow \text{rem}(S, v) \Rightarrow \{T\sigma\}
\]

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

**Lemma 5.** Given \( \Gamma, \Gamma' \in S(\Sigma) \) two structures such that \( \Gamma' \vdash \{T\sigma\} \), and \( S \in \Gamma \) as a set, then \( \Gamma \vdash wp(\text{rem}(S, v), \{T\sigma\}) \) if and only if \( \Gamma \vdash \exists X \left[ X \setminus S \not\in \Sigma \right] \Rightarrow \left( T \land \left\langle X \left( \sigma \setminus S \not\in \Sigma \right) \right) \Rightarrow \text{rem}(S, v) \Rightarrow \{T\sigma\} \).

**Proof.** The proof is based on the following equivalence:

\[
\exists X \left[ X \setminus S \not\in \Sigma \right] \Rightarrow \sigma \land \left\langle X \left( \sigma \setminus S \not\in \Sigma \right) \right] \Rightarrow \{T\sigma\}
\]

Let \( \Gamma \vdash wp(\text{rem}(S, v), \{T\sigma\}) \). Then, by Lemma 2 and since \( S \in \Gamma \), \( \Gamma' = \text{rem}(S, v, \Gamma) \) is also a structure and \( \Gamma' \vdash \{T\sigma\} \). Then \( \Gamma' \vdash \left\langle X \left( \sigma \setminus S \not\in \Sigma \right) \right] \Rightarrow \{T\sigma\} \).

\[
\Gamma \quin\exists X \left[ X \setminus S \not\in \Sigma \right] \Rightarrow \left\langle X \left( \sigma \setminus S \not\in \Sigma \right) \right] \Rightarrow \{T\sigma\}
\]

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

\[
\{T = S \land \sigma \land \exists v \neq 0 \lor \{T\sigma\} \Rightarrow \text{new}(S, v) \Rightarrow \{T\sigma\} \}
\]

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 \( wp(\text{new}(S, v), \varphi) \) the largest set of structures \( \Gamma \in S(\Sigma) \) which, under the transformation \( \text{new}(S, v) \), lead to a structure satisfying \( \varphi \). We need to assume the necessary and sufficient condition from Lemma 2 in order to ensure that the result of \( \text{new}(S, v) \) is a consistent storeless structure.

**Lemma 6.** Given \( \Gamma, \Gamma' \in S(\Sigma) \) two structures such that \( \Gamma \vdash \{T\sigma\} \), and \( S \in \Gamma \) as a set such that for all \( Y \in \Gamma' \), either \( S \cap Y = \emptyset \) or \( Y = S \), then \( \Gamma \vdash wp(\text{new}(S, v), \{T\sigma\}) \) if and only if \( \Gamma \vdash \left( T = S \land \sigma \land \exists v \neq 0 \lor \{T\sigma\} \right) \).

**Proof.** “\( \Rightarrow \)” Let \( \Gamma \vdash wp(\text{new}(S, v), \{T\sigma\}) \) be a storeless structure. Then, by Lemma 2 and since \( S \in \Gamma \), \( \Gamma' = \text{rem}(S, v, \Gamma) \) is also a structure and \( \Gamma' \vdash \{T\sigma\} \). Then \( \Gamma' \vdash \left\langle X \left( \sigma \setminus S \not\in \Sigma \right) \right] \Rightarrow \{T\sigma\} \).

**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).

\[
\exists X \left[ \chi(X) = U \land \bigvee_{i=1}^{\lambda} \psi_i(X) \right] \Rightarrow \text{add}(S, v, T) \Rightarrow \{U\sigma\}
\]

where

\[
\psi_i(X) \triangleq S(\neg T^{-1}X)^{v} \land \neg \chi(X) \land \neg \psi_i(X) \triangleq S(\neg T^{-1}X)^{v} \land \neg \chi(X)
\]

The soundness and completeness proof is done in a similar way, with \( wp(\text{add}(S, v, T), \varphi) \) 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 \( \Gamma, \Gamma' \in S(\Sigma) \) two structures such that \( \Gamma \vdash \{U\sigma\} \), and \( S, T \in \Gamma \) as two sets such that, for all \( Y \in \Gamma' \), either \( S \cap Y = \emptyset \) or \( Y = T \), then \( \Gamma \vdash wp(\text{add}(S, v, T), \{U\sigma\}) \) if only if \( \Gamma \vdash \exists X \left[ \chi(X) = U \land \bigvee_{i=1}^{\lambda} \psi_i(X) \right] \).

**Proof.** The proof is based on the following equivalences, which result immediately from rule (12). For any \( X \not\in \Sigma \) such that \( \chi(X) = U \):

\[
U \cap \sigma = \neg S(\neg T^{-1}X)^{v} \land \neg \chi(X) \land \neg U \cap \sigma = \psi_i(X) \lor \psi_i(X)
\]
"⇒" Let $Γ \in \text{up}(\text{add}(S, v, T), \{U\})$ be a storeless structure such that for all $Y \in Γ$, either $Sv \cap Y = \emptyset$ or $Y = T$. If we denote $Γ' = \text{add}(S, v, T, Γ)$, by Lemma 4, we have $Γ' \in S(Γ)$ and $Γ' \models \{U\}$. So $U \in Γ'$ and, by rule (12), $χ(X) = U$ for some $X \in Γ'$. Also, by (24) we have $ψ_1(X) \lor ψ_2(X)$. "⇐" Assume that $Γ \models \exists X [χ(X) = U \land \lor_{i=1,2} ψ_i(X)]$ and let $Γ' = \text{add}(S, v, T, Γ)$. If $ψ_i(X)$, $i = 1, 2$ holds on $Γ$ we have $X \in Γ'$. By rule (12) we have then $U \in Γ'$. Applying (24) we obtain $Γ' \models \{U\}$. □

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., $\Gamma$ always exists. This observation leads to the fact that the transition relation $\rightarrow$ defined by the rules in Figure 5 is both total and deterministic. According to the previous discussion, the weakest precondition predicate transformer $\text{pre}$ 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 $\text{op}$ be a primitive operation i.e., one of $\text{add}$, $\text{new}$ and $\text{add}$ provided with some sound parameters $S$, $v$ and $T$, then we denote by $\text{pre}(φ, ψ)$ the formula obtained by recursively applying rules (20), (22) and (23) to $φ$. Notice that, if $φ$ is a pure assertion, we have $\text{pre}(φ, ψ) = φ$. 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 $ω \in \text{Stmt}^*$ let $\text{pre}(ω, φ)$ be the precondition formula defined by the rules in Figure 7 using the classical composition rule $\{\text{pre}(m, \text{pre}(n, φ))\}$. Example

**Theorem 4.** Given $Γ, Γ' \in S(2)$ two structures, $ω \in \text{Stmt}^*$ a statement and $φ$ an AL formula such that $Γ' \models φ$, then $Γ \models \text{pre}(ω, φ)$ if and only if $Γ \models \text{pre}(ω, φ)$.

**Proof.** By induction on the structure of $φ$. □

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 $\text{pre}(ω, φ) \iff \neg \text{pre}(ω, \neg φ)$ does not hold any longer. Consequently we also lose the fact that $\text{pre}(ω, φ_1 \lor φ_2) \iff \text{pre}(ω, φ_1) \lor \text{pre}(ω, φ_2)$ and the same for existential quantifier. However, $\text{pre}(ω, φ_1 \lor φ_2) \iff \text{pre}(ω, φ_1) \lor \text{pre}(ω, φ_2)$ still holds, and similar for the existential quantifier. This results in a loss of completeness of weakest preconditions. Notice that, if we still replace $\text{pre}(ω, φ_1 \lor φ_2)$ by $\text{pre}(ω, φ_1) \lor \text{pre}(ω, φ_2)$ 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 $\text{pre}(ω, φ)$ by $\neg\text{pre}(ω, φ)$ 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 $\text{nclist}(i, n) \land \text{nclist}(j, n) \land \text{nsbared}(i, j, n)$ is an invariant of the while loop. Since we deal with total correctness, we shall assume a side condition of the form $∃I [\{T\}]$ 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 j ≠ null do
                  k := 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 $P = Q$ and $\{Q\} \subseteq \{R\}$ then $\{P\} \subseteq \{R\}$. 
\{nlist(i,n) \land nlist(j,n) \land nshared(i,j,n)\} \\
\Rightarrow \{nlist(i,n) \land nlist(j,n) \land nshared(i,j,n) \land \\
\forall i,j \left( (j \mathrm{f} \land (j \Rightarrow j \leq i \land n') = 0) \Rightarrow \right. \\
\left. k := n \right\} \\
\{nlist(k,n) \land nlist(j,n) \land nshared(k,j,n) \land \\
\forall i,j \left( (j \mathrm{f} \land (j \Rightarrow j \leq k \land n') = 0) \Rightarrow \right. \\
\left. i := j \right\} \\
\{nlist(k,n) \land nlist(i,n) \land nshared(k,i,n) \} \\
\Rightarrow \{nlist(i,n) \land nlist(j,n) \land 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 η-paths between the node containing j and the node containing i, or else the assignment \texttt{i.n := j} would generate a cycle, invalidating \texttt{nlist}(i,n). This precondition is however implied by \texttt{nshared}(i,j,n) and disappears when we apply the confluence 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 \texttt{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 \texttt{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 \texttt{AL} assertions as abstract domain, in the spirit of shape analysis [25].

7. REFERENCES


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 $\neg(U)\sigma = \neg(in(U)) \land \sigma \cap V = \emptyset$. 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 preconditions of a disjunctive formula $\neg(in(\phi_1 \lor \phi_2))$ $\Leftrightarrow$ $\neg(in(\phi_1)) \lor \neg(in(\phi_2))$. Disjunctive formulas are obtained, for instance, by applications of the add weakest precondition rule (23).

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

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

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

B. DERIVATION EXAMPLE

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

$$\exists K \ [(K)k \land \text{pre}(\text{rem}(e, i), \text{pre}(\text{add}(e, i, K), X)i))]$$

$$i := k$$

$$\{X(i)\}$$

Next, we calculate $\text{pre}(\text{add}(e, i, K), X)i) = \exists X'[\chi(X') = X \land i(K^{-1}X') \cap i = \emptyset \land (X')i \lor\exists X'[\chi(X') = X \land i(K^{-1}X') \cap i \neq \emptyset \land (X')i = X']$.

Applying the rule for $\text{rem}$ to the first disjunct we obtain, for some $X'$ and $X''$:

$$X'' \setminus (\exists^* i = X' \land \chi(X') = X \land i(K^{-1}X') \cap i = \emptyset \land (X')i$$

$$\exists X'[\chi(X') = X \land i(K^{-1}X') \cap i \neq \emptyset \land (X')i = X']$$

Notice that the last conjunct is false since $i \setminus (\exists^* i = X' \land \chi(X') = X \land i(K^{-1}X') \cap i = \emptyset$ and $X'' \cap \emptyset = \emptyset$. Hence we are left with the following term, obtained by application of the $\text{rem}$ rule to the second disjunct above:

$$X'' \setminus (\exists^* i = X' \land \chi(X') = X \land i(K^{-1}X') \cap i \neq \emptyset \land (X')i = X']$$

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

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