# Safety and Liveness in Concurrent Pointer Programs

Dino Distefano[1], Joost-Pieter Katoen[2,3] and Arend Rensink[3]

[1] Dept. of Computer Science, Queen Mary, University of London, United Kingdom
[2] Software Modeling and Verification Group, RWTH Aachen, Germany
[3] Formal Methods and Tools, University of Twente, The Netherlands

**Abstract.** The incorrect use of pointers is one of the most common source of software errors. Concurrency has a similar characteristic. Proving the correctness of concurrent pointer manipulating programs, let alone algorithmically, is a highly non-trivial task. This paper proposes an automated verification technique for concurrent programs that manipulate linked lists. Key issues of our approach are: automata (with fairness constraints), heap abstractions that are tailored to the program and property to be checked, first-order temporal logic, and a tableau-based model-checking algorithm.

## 1 Introduction

Pointers are an indispensable part of virtually all imperative programming languages, be it implicitly (like in Java or "pure" object-oriented languages) or explicitly (like in the C family of languages). However, programming with pointers is known to be error-prone, with potential pitfalls such as dereferencing null pointers and the creation of memory leaks. This is aggravated by aliasing, which may easily give rise to unwanted side-effects because apparently unaffected variables may be modified by changing a shared memory cell — the so-called *complexity of pointer swing.* The analysis of pointer programs has been a topic of continuous research interest since the early seventies [10, 15]. The purpose of this research is twofold: to assess the correctness of pointer programs, or to identify the potential values of pointers at compile time so as to allow more efficient memory management strategies and the use of code optimization.

The problems of pointer programming become even more pressing in a concurrent setting where memory is *shared* among threads. Since the mainstream object-oriented languages all offer shared-memory concurrency, this setting is in fact quite realistic. Concurrent systems are difficult enough to analyze in the absence of pointers; the study of this area has given rise to techniques such as process algebra [30, 6], temporal logic [35] and comparative concurrency theory [23]. Techniques for analyzing programs that feature both concurrency and pointers are scarce indeed.

*Properties of pointer programs.* Alias analysis, i.e., checking whether pairs of pointers can be aliases, has received much attention (see, e.g., [13, 26]) initially.

[16] introduced and provided algorithms to check the class of so-called position-dependent alias properties, such as "the $n$-th cell of $v$'s list is aliased to the $m$-th cell of list $w$". Recently, extensions of predicate calculus to reason about pointer programs have become en vogue: e.g., *BI* [24], separation logic [37], pointer assertion logic (*PAL*) [25], alias logic [8, 9], local shape logic [36] and extensions of spatial logic [11]. These approaches are almost all focused on verifying pre- and postconditions in a Hoare-style manner.

Since our interest is in concurrent (object-oriented) programs and in expressing properties over dynamically evolving pointer (i.e., object reference) structures, we use first-order linear-time temporal logic (LTL) as a basis and extend it with pointer assertions on single-reference structures, such as aliasing, position-dependent aliasing, as well as predicates to reason about the birth and death of cells. This results in an extension of propositional logic, which we call *NTL* (Navigation Temporal Logic), similar in nature to that proposed in Evolution Temporal Logic (*ETL*) [40] — see below for a more detailed comparison. The important distinguishing feature of these logics with respect to "plain old" propositional logic is that *quantification occurs outside the temporal modalities*; in other words, we can reason about the evolution of entities over time. This type of logic is known as *quantified modal logic*; see, e.g., [21, 3]. This is in contrast to *PAL*, which contains similar pointer assertions as *NTL* (and goes beyond lists), but has neither primitives for the birth and death of cells nor temporal operators.

In the semantics of *NTL* (in contrast to *ETL*, which uses 3-valued logical structures) we follow the traditional automata-based approach: models of *NTL* are infinite runs that are accepted by Büchi automata where states are equipped with a representation of the heap. (In terms of quantified modal logic, our models have variable domains and are non-rigidly designating.) Evolving heaps have been lately used to model mobile computations. In that view *NTL* combines both spatial and temporal features as the ambient logic introduced in [11]. In fact in [17] one of the authors has shown how to use the *NTL* model to analyze mobile ambients.

*Heap abstraction.* Probably the most important issue in analyzing pointer programs is the choice of an appropriate representation of the heap. As the number of memory cells for a program is not known a priori and in general is undecidable, a concrete representation is inadequate. Analysis techniques for pointer programs therefore typically use abstract representations of heaps such as, e.g., location sets [39] (that only distinguish between single and multiple cells), $k$-limiting paths [26] (allowing up to $k$ distinct cells for some fixed $k$), or summary nodes [38] in shape graphs. This paper uses an abstract representation that is tailored to unbounded linked list structures. The novelty of our abstraction is its parameterization in the pointer program as well as in the formula. Cells that represent up to $M$ elements, where $M$ is a formula-dependent constant, are exact whereas unbounded cells (akin to summary nodes) represent longer lists. The crux of our abstraction is that it guarantees each unbounded cell to be preceded by a chain of at least $L$ exact cells, where $L$ is a program-dependent constant.

Parameters $L$ and $M$ depend on the longest pointer dereferencing in the program and formula, respectively. In contrast with the $k$-limiting approach, where an adequate general recipe to determine $k$ is lacking, we show how (minimal bounds on) the parameters $L$ and $M$ can be determined by a simple static analysis.

*Pointer program analysis.* Standard type-checking is not expressive enough to establish properties of pointer programs such as (the absence of) memory leaks and dereferencing null pointers. Instead, existing techniques for analyzing pointer programs include abstract interpretation [16], deduction techniques [8, 22, 24, 33, 37, 25, 32], design by derivation *a la* Dijkstra [28], and shape analysis [38], or combinations of these techniques.

We pursue a fully automated verification technique, and for that reason we base our approach on model checking. Our model-checking algorithm is a non-trivial extension of the tableau-based algorithm for LTL [27], tailored to the variable-domain models described above. For a given *NTL*-formula $\Phi$, this algorithm is able to check whether $\Phi$ is valid in the automaton-model of the concurrent pointer program at hand. The algorithm, like in any approach based on abstraction, is approximative: in our case this means that it suffers from false negatives, i.e., a verification may wrongly conclude that the program refutes a formula. In such a case, however, diagnostic information can be provided (unlike *ETL*, and as for *PAL*) that may be used for further analysis. Besides, by incrementing the parameters $M$ and $L$, a more concrete model is obtained that is guaranteed to be a correct refinement of the (too coarse) abstract representation. This contrasts with the *ETL* approach where manually-provided instrumentation predicates are needed. As opposed to the *PAL* approach, which is fully automated only for sequential loop-free programs, our technique is fully automated for concurrent pointer programs that may include loops.

*Main contributions.* Summarizing, the main contributions of this paper are:

1. A quantified temporal logic (with some second-order features) that contains pointer assertions as well as predicates referring to the birth or death of memory cells;
2. An automaton-based model for pointer programs where states are abstract heap structures and transitions represent the dynamic evolving of these heaps; the model deals finitely with unbounded allocations.
3. A program analysis that automatically derives an over-approximation of the invariant of concurrent programs manipulating lists. This analysis is sound and it is guaranteed to terminate.
4. A control on the degree of *concreteness* of abstract heap structures, in the form of two parameters that are obtained by a straightforward static analysis of the program and formula at hand. On incrementing these parameters, refined heap structures are automatically obtained. Vice-versa, by decrementing them more abstract models are derived. Hence the process of abstraction-refinement of the analysis is reduced to only tuning these two numeric parameters.

5. A model checking algorithm to verify safety and liveness properties (expressed by formulae in our logic) against abstract representations of pointer programs.

This results in a push-button technique: given a program and a temporal logic property, the abstract automaton as well as the verification result for the property are determined completely algorithmically. Moreover, to our knowledge, we are the first to develop model-checking techniques for (possibly) unbounded evolving heaps of the kind described above. (Recently, regular model checking has been applied to check properties of linked lists [7])

Our current approach deals with single outgoing pointers only. This still allows us to consider many interesting structures such as acyclic, cyclic, shared and unbounded lists (as in [28] and [16]), as well as hierarchies (by back-pointers). Besides, several *resource managers* such as memory managers only work with lists [34]. Moreover, several kernel routines and device drivers uses lists. Our abstract heap structures can also model mobile ambients [17].

*Related work.* Above we have already mentioned many sources of related work. Two of them, however, deserve a more detailed discussion: *shape analysis* and *separation logic.*

In [38], a framework for the generation of a family of shape analysis algorithms based on 3-valued logic and abstract interpretation is presented. This very general framework can be instantiated in different ways to handle different kinds of data structures at different levels of precision and efficiency.

The similarity between the analysis in [38] and ours is mostly in the use of summary nodes in order to obtain finite states representation of the invariant of the program. However, our summaries are only used (and tailored) to abstract lists whereas in [38] they can be more general. In fact, since in [38] states are represented by 3-valued logical structures, the abstraction is done by the partitioning induced by the predicate values (canonical abstraction). In contrast, our abstraction is technically implemented by means of morphisms which keep a strong correspondence between the abstract heap and the concrete ones it represents.

Among the differences between the two approaches, we have that [38] gives a collecting semantics of the program. We use an automata semantics which allows us to apply temporal reasoning and verify a wide range of safety and liveness properties. Also, the framework of [38] makes use of instrumentation predicates to refine the analysis whereas the refinement in our case is done by tuning two numerical parameters. Moreover using morphisms, the soundness of the new refined model is automatically guaranteed and therefore there is no need to provide a proof of the equivalence for the two models.

The closest extension of [38] to our work is the aforementioned [40] on a first-order modal (temporal) logic (called *ETL*) for allocation and deallocation of objects and threads as well as for the specification of properties related to the evolution of the heap. Although the aims of that paper and ours are surprisingly close (for example in the kind of properties expressible in *NTL* and *ETL*), the technical machinery has those differences mentioned above between our work and the setting of [38]. Moreover, [40] uses a trace semantics where each trace is encoded by first-order logical structure. Formulae of *ETL* are then translated in first-order logic with transitive closure for the evaluation on a trace. We use Büchi automata to generate traces and verify *NTL* by an extension of the LTL model-checking algorithm.

Separation logic [37, 24, 34] is an extension of Hoare logic able to prove heap-manipulating programs in a concise and modular manner. At the core of separation logic there is a new operator $*$ called *separating conjunction.* The formula $P * Q$ holds if $P$ and $Q$ hold in disjoint parts of memory. The $*$ operator stands at the foundation of local reasoning in separation logic: it allows one to focus only on the cells that are accessed by the program without the need to keep track of possible aliases. Lately a lot of attention has been devoted in the design of decision procedures and tools for program analysis that uses separation logic as an effective model [4, 5].

Although separation logic uses the random access memory model, it seems that it would be possible to give a graph-based semantics for the logic. Interestingly, for subsets of separation logic working only on lists, many features, of this model would be very similar to the heaps we have introduced in this paper. For example, our unbounded entities would correspond to the predicate $\mathsf{listseg}(x, y)$ indicating a pure list segment from $x$ to $y$. It would be interesting to see if the *frame rule* of separation logic, which allows modular reasoning about heap manipulating programs, can be proved sound for such graph model.

*Outline of the paper.* This paper is, in a sense, a companion to [18] and a summary (and partial revision) of [19], where we presented the technical details of the automata, logic and (to some degree) the model checking algorithm. Here we focus more on the usability aspects. We present the concurrent pointer language in Section 2 and discuss a number of examples. Section 3 presents a concrete semantics for the language. In Section 4 we define the operational semantics on the basis of the abstract automata described above; in Section 5 we introduce the logic and its semantics, and discuss it on the basis of the examples given in Section 2. We also (quite briefly) discuss the principles of the model checking algorithm, in Section 6. Details of the model checking algorithm and all proofs can be found in [19].

## 2 Concurrent Pointer-Manipulating Programs

This section introduces a simple concurrent programming language dealing with pointer structures. It incorporates means to create and destroy heap cells (referred to by pointers), and operations to manipulate them. A concurrent producer-consumer problem and an in-place reversal program are used to illustrate the kind of programs that can be written. In the discussion of these programs, some relevant temporal properties will be introduced. Later on in the paper, the formal specification and verification of these properties is treated.

## 2.1 Programming language

Let $PV$ be a set of program variables with $v, v_i \in PV$. Each program variable is assumed to denote a memory cell, where the constant *nil* is treated as a special cell. A program variable is said to be undefined in case it is not pointing to any cell. The syntax of programs is given by the following grammar:

$$
\begin{aligned}
p &::= \mathbf{var}\ v_1, \ldots, v_n : (s_1 \parallel \cdots \parallel s_k) \\
s &::= \mathsf{new}(\ell) \;\Big|\; \mathsf{dispose}(\alpha) \;\Big|\; \ell := \alpha \;\Big|\; \mathsf{skip} \;\Big|\; s; s \;\Big|\; \mathbf{if}\ (b)\{\, s \,\}\{\, s \,\} \;\Big| \\
&\qquad \mathbf{while}\ (b)\{\, s \,\} \;\Big|\; \langle s \rangle \;\Big|\; \mathsf{error} \\
\alpha &::= nil \;\Big|\; v \;\Big|\; \alpha{\uparrow} \\
\ell &::= v \;\Big|\; \ell{\uparrow} \\
b &::= \alpha = \alpha \;\Big|\; \mathsf{undef}(\alpha) \;\Big|\; b \vee b \;\Big|\; \neg b
\end{aligned}
$$

Thus, a program $p$ is a parallel composition of a finite number of statements preceded by the declaration of a finite number of global variables. Statements have the following intuitive interpretation.

- new($\ell$) creates (i.e., allocates) a new cell that will be referred to by $\ell$. The old value of $\ell$ is lost. Thus, if $\ell$ is the only pointer to cell $e$, say, then after the execution of new($\ell$), $e$ has become "unreachable". In this case, $e$ is automatically garbage collected together with the entities that are only reachable from $e$.
- dispose($\alpha$) destroys (i.e., deallocates) the cell associated to $\alpha$, and makes $\alpha$ and every other pointer referring to it undefined. For the sake of simplicity, new and dispose create, respectively destroy, a single entity only; generalizations in which several entities are considered simultaneously can be added in a straightforward manner.
- The assignment $\ell := \alpha$ assigns a reference to the cell denoted to by $\alpha$ to $\ell$. (Note that *nil* cannot occur as left-hand side of an assignment.) Again, the cell that $\ell$ was referring to might become unreferenced, in which case it is removed by garbage collection.
- Sequential composition, while, skip and if have the standard interpretation. The statement $\langle s \rangle$ denotes an atomic region, i.e., all statements in $s$ are executed atomically, without possible interference of any other concurrent statement.
- $\alpha$ stands for a *pointer expression* whereas $\ell$ stands for a *location*. The suffix $\uparrow$ in both cases expresses dereferencing, or following the single outgoing pointer. We denote $x{\uparrow}^0 = x$ and $x{\uparrow}^{n+1} = (x{\uparrow}^n){\uparrow}$.
- The expression undef($\alpha$) yields true if and only if $\alpha$ is undefined (which can for instance happen as a consequence of dispose($\beta$) if originally $\beta = \alpha$). Obviously, this is different from testing for $\alpha = nil$. The capability of testing for undefinedness within the language can be useful if we want to express behavior on the level of system programs.

Statements containing meaningless but legal expressions, such as dispose($nil$) and $nil\uparrow$, will result in a run-time error (as defined by our semantics later on). The halting of a statement due to such error is indicated by the construct error. This construct is thus a semantical one, and cannot be part of any program.

## 2.2 Some example programs

*Producer-consumer programs.* Consider the concurrent producer-consumer problem that consists of three concurrent processes. The producer process repeatedly generates new items by allocating new memory cells referred to by the global program variable $p$. It does so only when $p$ is undefined. A one-place buffer process copies the memory cell referred to by $p$ (if any) and makes $p$ undefined. As soon as an item is available in the buffer, the consumer process is able to empty the buffer by disposing the memory cell. Typical properties that the producer-consumer program should satisfy are:

- absence of memory leaks, i.e., any produced item is eventually consumed
- first-in first-out property, i.e., items are consumed in the order of production
- unboundedness of the number of produced items

The first and last are typical liveness property, whereas the second is a safety property.

To show the intricacy of the producer-consumer problem, we present several programs that are slight variants of each other, and discuss their properties. A first producer-consumer program that realizes the sketched approach is:

```
var c, p, w :                                              // program variables
(   while (true) if (undef(p)) {new(p); }                  // producer
 || while (true) if (¬undef(p)) {c := p; p := w; }         // buffer
 || while (true) if (¬undef(c)) {dispose(c); }             // consumer
)
```

This first program clearly suffers from a memory leak, as it allows produced cells to be never consumed. This can be expressed by "possibly, a produced entity is never referred to by $c$". This stems from the fact that the producer can put a new item in the buffer before the consumer retrieves the previous one. The following variant avoids this problem by exploiting the auxiliary variable $w$ (which was used before just to make $p$ undefined). The buffer process thus becomes:

```
while (true) if (¬undef(p)) {w := p; p := c; c := w; }
```

whereas the producer and consumer processes remain as before. This program indeed has no memory leak — provided that the consumer process is scheduled infinitely often — but violates the order-preservation property; i.e., items may be consumed in a different order than they are produced. This occurs for instance in Fig. 1, which represents an example run of the program.

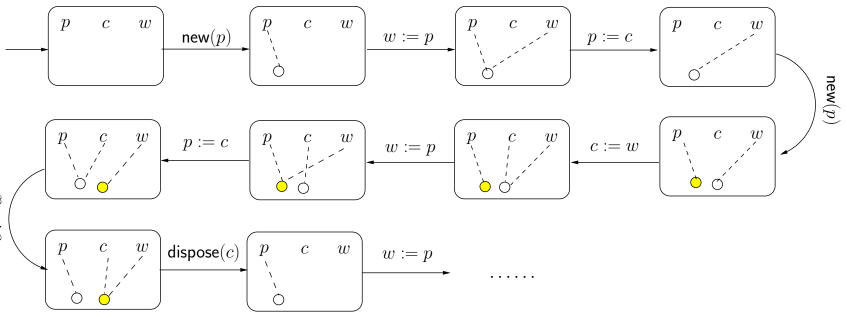


**Fig. 1.** The order of consumption $\neq$ the order of production

To overcome this problem, the guard of the buffer process is strengthened such that the producer is only allowed to put a new item into the buffer whenever the previous item has been retrieved. This yields the following buffer process:

$$\textbf{while } (\mathit{true}) \textbf{ if } (\neg \mathsf{undef}(p) \wedge \mathsf{undef}(c)) \; \{w := p; p := c; c := w; \}$$

The producer and consumer process are as before. It can be shown that this program indeed satisfies all properties: it guarantees that memory leaks cannot occur (assuming process fairness), the order of production is preserved, and an unbounded number of items is produced.

Although the discussed programs can in the course of time produce an unbounded number of items, the buffer capacity is still finite. This is no longer valid for the following variant. Rather than modeling the buffer as a separate process, we consider the buffer to be realized as a global linked list of unbounded length. The producer adds entities to the tail *tl* of the buffer, whereas the consumer process removes and consumes them from the head *hd* of the buffer.

```
var $hd, tl, t$ :
(   new($tl$); $hd := tl$; while ($true$) {new($tl\uparrow$); $tl := tl\uparrow$ }          // producer
 || while ($true$) if ($hd \neq tl$) {$t := hd$; $hd := hd\uparrow$; dispose($t$) } // consumer
)
```

In addition to the previously mentioned properties, it is desirable that during the execution of this program, the tail of the buffer never gets disconnected from the head.

*In-place list reversal.* As a final example we consider a classical sequential list-manipulating problem, viz. reversing the direction of a list. We show two solutions, one of which is actually incorrect. Both programs try to establish reversal in a destructive (or so-called *in-place*) manner as they reuse the cells of the original list, as initially pointed to by program variable $v$, to built the reversed list. Properties of interest for this problem include, for instance:

– $v$'s list will be (and remains to be) reversed;
– none of the elements in $v$'s list will ever be deleted;

– $v$ and $w$ always point to distinct lists (heap non-interference).

Here, $w$ is an auxiliary variable that is used in the construction of the reversed list. The following program is taken from [2], but violates the first property.

```
var $v, w, t, z$ :
if $(v \neq nil)$ {
  $t := v\uparrow; w := nil;$
  while $(t \neq nil)$ {
    $z := t\uparrow; v\uparrow := w; w := v; v := t; t := z$
  }
}
```

The problem in this erroneous program is that one pointer is missing in the reversed list. Before continue reading, the reader is invited to find the error in the program.

The following list-reversal program (see, e.g., [8, 37, 38]) reverses the list in a correct manner, i.e., this program satisfies the three aforementioned properties:

```
var $v, w, t$ :
$w := nil;$
while $(v \neq nil)$ {
  $t := w; w := v; v := v\uparrow; w\uparrow := t$
}
```

### 2.3 The topic of this paper

To check properties like the ones in the previous examples in a *fully automated* manner is the challenge that is faced in this paper. We advocate an automata-based model-checking approach in which states are equipped with (abstract) heap representations. As property-specification language we propose to use a first-order variant of linear temporal logic. Before explaining the heap abstraction mechanism, we provide the concrete (and infinite-state) semantics of our example programming language. This is characterized by the fact that each cell and pointer is represented explicitly.

## 3 Concrete Semantics

We assume a universe of entities, *Ent*, including a distinguished element *nil*, used to represent a canonical entity without outgoing references. We let $PV \subseteq Ent$, i.e., program variables are assumed to correspond to special entities that exist throughout the entire computation.

*Configurations.* Automata will be used as semantical model for our programming language. States (called configurations) of these automata are equipped with information about the current entities, their pointer structure, and the current set of fresh entities.

**Definition 1 (configuration).** *A* configuration *is a tuple* $c = \langle E, \prec, N \rangle$ *such that:*

- $E \subseteq Ent$ *is a finite set of entities, with* $nil \in E$.
- $\prec \subseteq E \times E$ *is a binary relation over* $E$*, such that:*

$$outdegree_{\prec}(e) \leqslant 1 \text{ for all } e \in E \quad \text{and} \quad outdegree_{\prec}(nil) = 0$$

- $N \subseteq E$ *is the set of fresh entities*

*$c$ is called* reachable *if for all* $e \in E$*, there is some* $e' \in PV \cap E$ *such that* $e' \prec^* e$ *(where* $\prec^*$ *is the reflexive and transitive closure of* $\prec$*).*

A configuration is used to model the heap of a program. Note that in general these concrete heaps can grow unboundedly. The only data structure allowed is a cell with at most a single pointer to another cell. The cells are modeled by entities and the pointers by the binary relation $\prec$. Note the restriction on the $\prec$-outdegree, which implies that any entity has at most one $\prec$-successor. The derived partial function $succ : E \rightharpoonup E$ is defined by:

$$succ(e) = e' \quad \text{if} \quad e \prec e' \quad .$$

*Example 1.* In Fig. 1, the configurations are depicted as ovals, program variables stand for the entities representing them, and the dashed lines between variables and entities represent the pointers from $e_v$ to $succ(e_v)$. The entity $nil$ is not depicted in these configurations. If in cells the outgoing pointer is not depicted then it is dangling. The fresh entities in a configuration are the entities that are absent in the previous configuration. They typically arise as a result of executing the new statement.

*Interpreting navigation expressions.* The semantics of navigation expression $\alpha$ in configuration $c = \langle E, \prec, N \rangle$ is given by:

$$\begin{aligned} [\![nil]\!]_c^{\mathsf{exp}} &= nil \\ [\![v]\!]_c^{\mathsf{exp}} &= succ(v) \\ [\![\alpha\uparrow]\!]_c^{\mathsf{exp}} &= succ\left([\![\alpha]\!]_c^{\mathsf{exp}}\right) \end{aligned}$$

where $succ$ is assumed to be strict, i.e., $succ(\bot) = \bot$. We omit the subscript $c$ from $[\![\,]\!]_c^{\mathsf{exp}}$ in case the configuration is clear from the context. The semantics of left-hand sides of assignments is defined as:

$$\begin{aligned} [\![v]\!]_c^{\mathsf{loc}} &= v \\ [\![\ell\uparrow]\!]_c^{\mathsf{loc}} &= succ\left([\![\ell]\!]_c^{\mathsf{loc}}\right) \end{aligned}$$

Note that $[\![v]\!]^{\mathsf{loc}}$ equals the entity denoting $v$ in case $v$ occurs as left-hand side of an assignment (or as argument of new), whereas $[\![v]\!]^{\mathsf{exp}}$ is the cell referred to by $v$ whenever $v$ occurs as right-hand side (or as argument of dispose).

*Heap manipulations.* The following operations on configurations are useful to define the operational semantics of operations such as new, dispose and assignment. All operations manipulate the heap, and yield a new heap that is obtained by either adding or deleting entities, or by changing pointers. Assume w.l.o.g. that *Ent* is totally ordered by some arbitrary natural ordering; this is convenient for selecting a fresh entity in a deterministic way. The following operations require $[\![\ell]\!]^{\mathsf{loc}}$ and $[\![\alpha]\!]^{\mathsf{exp}}$ to be different from $\perp$ and *nil*.

– The operation $\mathit{add}(c, \ell)$ extends the configuration $c = \langle E, \prec, N\rangle$ with a fresh entity $e$ referred to by the expression $\ell$:

$$\begin{aligned}\mathit{add}(c,\ell) = \langle E \cup \{e\}, \prec', \{e\}\rangle \text{ with } & e = \min(\mathit{Ent} \setminus E)\\ & \prec' = \prec \setminus \{([\![\ell]\!]^{\mathsf{loc}}, [\![\ell]\!]^{\mathsf{exp}})\} \cup \{([\![\ell]\!]^{\mathsf{loc}}, e)\}\end{aligned}$$

– The operation $\mathit{cancel}(c, \alpha)$ deletes the entity denoted by the navigation expression $\alpha$ from the configuration $c$:

$$\mathit{cancel}(c, \alpha) = \langle E', \prec \cap (E' \times E'), \varnothing\rangle, \text{ with } E' = E \setminus \{[\![\alpha]\!]^{\mathsf{exp}}\}$$

Note that in the resulting configuration, every pointer to $[\![\alpha]\!]^{\mathsf{exp}}$ becomes undefined.

– Finally, the operation $\mathit{modify}(c, \ell, \alpha)$ changes the configuration $c$ such that the entity denoted by $\ell$ points to the entity referred to by $\alpha$:

$$\begin{aligned}\mathit{modify}(c, \ell, \alpha) = \langle E, \prec', \varnothing\rangle \text{ with } \prec' = & \prec \setminus \{([\![\ell]\!]^{\mathsf{loc}}, [\![\ell]\!]^{\mathsf{exp}})\}\\ & \cup \{([\![\ell]\!]^{\mathsf{loc}}, [\![\alpha]\!]^{\mathsf{exp}})\}\end{aligned}$$

Stated in words, the outgoing pointer of the entity denoted by $\ell$ is redirected to the entity denoted by $\alpha$.

The final operation on heap structures that is needed for the semantics is garbage collection. This is done by explicitly determining the entities in a configuration that are "reachable" (via the pointer structure) from some program variable. We define:

$$\mathit{gc}(c) = \langle E', \prec \cap (E' \times E'), N \cap E'\rangle \text{ with } E' = \{e \mid \exists e' \in \mathit{PV}.\, e' \prec^* e\}$$

Example applications of the heap manipulations are provided in Fig. 2.

*Pointer automata.* The semantics of the programming language is given by a pointer automaton, in fact an automaton that accepts infinite sequences of configurations according to a generalized Büchi acceptance condition. Each state in the pointer automaton is equipped with a concrete configuration that represents the current heap content.

**Definition 2 (pointer automaton).** *A* pointer automaton *$A$ is a tuple $(Q, \mathit{cf}, \rightarrow, I, \mathcal{F})$ where:*

– *$Q$ is a non-empty, denumerable set of states*

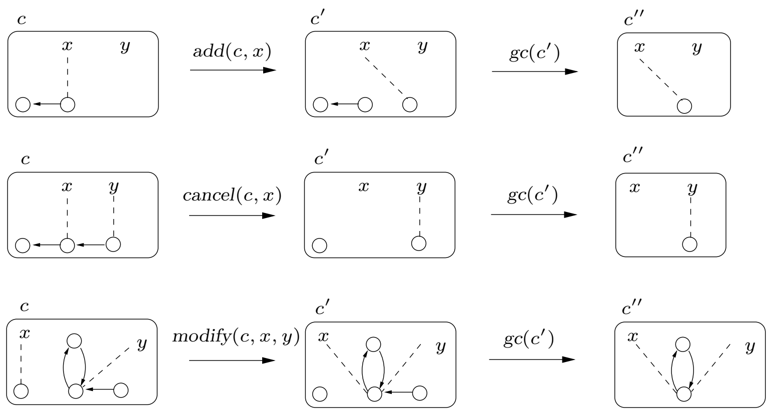


**Fig. 2.** Example heap manipulations

- *$cf\colon Q \to Cnf$ is a mapping associating a configuration with every state*
- *$\to \subseteq Q \times Q$ is a transition relation*
- *$I \subseteq Q$ is a set of initial states, and*
- *$\mathcal{F} \subseteq 2^Q$ is a generalized Büchi acceptance condition.*

We write $q \to q'$ instead of $(q, q') \in\ \to$. According to the generalized Büchi acceptance condition, $q_0 q_1 q_2 \cdots$ is an accepting *run* of $A$ if $q_i \to q_{i+1}$ for all $i \geqslant 0$, $q_0 \in I$ and $|\{i \mid q_i \in F\}| = \omega$ for all $F \in \mathcal{F}$. That is, each accept set $F \in \mathcal{F}$ needs to be visited infinitely often. Let *runs*$(A)$ denote the set of runs of $A$. Run $q_0 q_1 q_2 \cdots$ accepts the sequence of configurations $cf(q_0)\, cf(q_1)\, cf(q_2) \cdots$. The language $\mathcal{L}(A)$ denotes the set of configuration sequences that is accepted by some run of $A$.

Note that here and in the sequel we will implicitly interpret all configuration sequences *up to isomorphism*, where a sequence $c_0\, c_1\, c_2 \cdots$ is isomorphic to $c'_0\, c'_1\, c'_2 \cdots$ if each $c_i$ is isomorphic to $c'_i$, in the natural sense of having a bijective mapping $\psi_i : E_i \to E'_i$ that both preserves and reflects structure, and, moreover, $\psi_i(e) = \psi_{i+1}(e)$ for $e \in E_i \cap E_{i+1}$.

*Operational semantics.* Let *Par* denote the compound statements, i.e., the statements generated by $r ::= s \mid r \parallel s$ with $s$ a program statement as defined in Section 2. The compound statements constitute the states of the pointer automaton that will be associated to a program. We first provide the inference rules for those statements that might affect the heap structure, cf. Table 1. Any manipulation on an undefined navigation expression results in a run-time error, denoted by the process error. Any attempt to delete, create or assign a value to the constant *nil* fails too. These errors are considered to be local, i.e., the process attempting to execute these statements aborts, but this does not affect other concurrent processes. This will become clear from the rules for parallel composition.

The semantics of the other control structures is defined by the rules in Table 2. The rules for the alternative and sequential composition as well as for iteration are straightforward. Note that the boolean expression undef$(\alpha)$ yields true

$$\frac{[\![\ell]\!]^{\mathsf{loc}} \notin \{\bot, \mathit{nil}\}}{\mathsf{new}(\ell), c \longrightarrow \mathsf{skip}, gc \circ \mathit{add}(c, \ell)} \qquad \frac{[\![\ell]\!]^{\mathsf{loc}} \in \{\bot, \mathit{nil}\}}{\mathsf{new}(\ell), c \longrightarrow \mathsf{error}, c}$$

$$\frac{[\![\alpha]\!]^{\mathsf{exp}} \notin \{\bot, \mathit{nil}\}}{\mathsf{dispose}(\alpha), c \longrightarrow \mathsf{skip}, gc \circ \mathit{cancel}(c, \alpha)} \qquad \frac{[\![\alpha]\!]^{\mathsf{exp}} \in \{\bot, \mathit{nil}\}}{\mathsf{dispose}(\alpha), c \longrightarrow \mathsf{error}, c}$$

$$\frac{[\![\ell]\!]^{\mathsf{loc}} \notin \{\bot, \mathit{nil}\}}{\ell := \alpha, c \longrightarrow \mathsf{skip}, gc \circ \mathit{modify}(c, \ell, \alpha)} \qquad \frac{[\![\ell]\!]^{\mathsf{loc}} \in \{\bot, \mathit{nil}\}}{\ell := \alpha, c \longrightarrow \mathsf{error}, c}$$

**Table 1.** Operational rules for heap manipulations.

whenever $[\![\alpha]\!]^{\mathsf{exp}} = \bot$, and false otherwise. The semantics of the other boolean expressions is standard (where equality is assumed to be strict) and is omitted here. The semantics of atomic regions are determined by three rules. On entering an atomic region, the process $s$ is marked as "being in control"; this is indicated by the prefix atomic. This mark is lost once the atomic region is left, or whenever an error occurs. Once marked as being atomic, the process has control and is allowed to complete its atomic region without any possible interference of any other process. This is established by the first two rules for parallel composition. Once all processes are finished or aborted, the program loops. This is established by the last inference rule, and is exploited to impose fairness constraints. As a result, all runs of any pointer program are infinite.

$$\frac{[\![b]\!]^{\mathsf{exp}} = \mathit{true}}{\mathbf{if}\ (b)\{s_1\}\{s_2\}, c \longrightarrow s_1, c} \qquad \frac{[\![b]\!]^{\mathsf{exp}} = \mathit{false}}{\mathbf{if}\ (b)\{s_1\}\{s_2\}, c \longrightarrow s_2, c}$$

$$\frac{s_1, c \longrightarrow\ s_1', c' \wedge s_1' \notin \{\mathsf{skip}, \mathsf{error}\}}{s_1\ ;\ s_2, c \longrightarrow s_1'\ ;\ s_2, c'} \qquad \frac{s_1, c \longrightarrow \mathsf{skip}, c'}{s_1\ ;\ s_2, c \longrightarrow s_2, c'} \qquad \frac{s_1, c \longrightarrow \mathsf{error}, c}{s_1\ ;\ s_2, c \longrightarrow \mathsf{error}, c}$$

$$\frac{}{\mathbf{while}\ (b)\{s\}, c \longrightarrow \mathbf{if}\ (b)\{\ s\ ;\ \mathbf{while}\ (b)\{s\}\ \}\{\ \mathsf{skip}\ \}, c} \qquad \frac{s, c \longrightarrow s', c'}{\langle s \rangle, c \longrightarrow \mathsf{atomic}\ s', c'}$$

$$\frac{s, c \longrightarrow s', c'\ \wedge\ s' \notin \{\mathsf{skip}, \mathsf{error}\}}{\mathsf{atomic}\ s, c \longrightarrow \mathsf{atomic}\ s', c'} \qquad \frac{s, c \longrightarrow s', c'\ \wedge\ s' \in \{\mathsf{skip}, \mathsf{error}\}}{\mathsf{atomic}\ s, c \longrightarrow s', c'}$$

$$\frac{s_j, c \longrightarrow s_j', c'\ \wedge\ s_j' \neq \mathsf{error}\ \wedge\ (\forall i \neq j.\, s_i \neq \mathsf{atomic}\ s_i')}{s_1 \parallel \cdots \parallel s_j \parallel \cdots \parallel s_k, c \longrightarrow s_1 \parallel \cdots \parallel s_j' \parallel \cdots \parallel s_k, c'}$$

$$\frac{s_j, c \longrightarrow \mathsf{error}, c\ \wedge\ (\forall i \neq j.\, s_i \neq \mathsf{atomic}\ s_i')}{s_1 \parallel \cdots \parallel s_j \parallel \cdots \parallel s_k, c \longrightarrow s_1 \parallel \cdots \parallel \mathsf{error} \parallel \cdots \parallel s_k, c}$$

$$\frac{\forall 0 < j \leqslant k.\, s_j \in \{\mathsf{skip}, \mathsf{error}\}}{s_1 \parallel \cdots \parallel s_k, c \longrightarrow s_1 \parallel \cdots \parallel s_k, c}$$

**Table 2.** Operational rules for the control structures.

**Definition 3.** *The concrete semantics of program*

$$p = \mathsf{decl}\ v_1, \ldots, v_n : (s_1 \parallel \cdots \parallel s_k)$$

*is the pointer automaton* $[\![p]\!]^{\mathsf{conc}} = (Q, cf, \rightarrow, I, \mathcal{F})$ *such that:*

- $Q \subseteq Par \times Cnf$ *with* $cf(r, c) = c$
- $\rightarrow\ \subseteq Q \times Q$ *is the smallest relation satisfying the rules in Table 1 and 2;*
- $I = \{(s_1 \parallel \cdots \parallel s_k, \langle\{v_1, \ldots, v_n\}, \varnothing, \varnothing\rangle)\}$
- $\mathcal{F} = \{\widehat{\mathcal{F}}_i \mid 0 \leqslant i < k\} \cup \{\widetilde{\mathcal{F}}_i \mid 0 \leqslant i < k\}$ *where:*

$$\widehat{\mathcal{F}}_i = \{(s'_1 \| \cdots \| s'_k, c) \in Q \mid s'_i = \mathsf{skip} \vee s'_i = \mathsf{error} \vee s'_i = \mathbf{while}(b)\{s\}; s''\}$$
$$\widetilde{\mathcal{F}}_i = \{(s'_1 \| \cdots \| s'_k, c) \in Q \mid s'_i = \mathsf{skip} \vee s'_i = \mathsf{error} \vee s'_i = s; \mathbf{while}(b)\{s\}; s''\}.$$

A few remarks are in order. For state $(r, c) \in Q$, $r$ is the compound statement to be executed and $c$ is a reachable (concrete) configuration, i.e., it only contains the entities reachable from some program variable in the program $p$. $[\![p]\!]^{\mathsf{conc}}$ has a single initial state $s_1 \parallel \cdots \parallel s_k$ together with a heap that initially contains a cell for each program variable only. The set of accept states for the $i$-th sequential component $s_i$ consists of all states in which the component $i$ has either terminated ($s_i = \mathsf{skip}$), aborted ($s_i = \mathsf{error}$), or is processing a loop (which could be infinite). Note that according to this acceptance condition, processes that consist of an infinite loop are executed in a fair manner. This applies, e.g., to both processes in the producer-consumer example.

*Example 2.* Consider the example programs provided in Section 2. It can be checked that Fig. 1 indeed is a possible run that is allowed by the semantics of the second producer-consumer program. The transition labels are provided for convenience only. The initial part of the (infinite-state) pointer automaton that is obtained for the producer-consumer program with the shared list is given in Fig. 3.

## 4 Heap Abstractions

The most obvious way to model pointer structures is to represent each entity and each pointer individually as we did in the previous section. For most programs, like, e.g., the producer/consumer program with the shared linked list, this will give rise to infinite pointer automata. To obtain more abstract (and compact) views of pointer structures, chains of cells will be aggregated and represented by one (or more) cells. We consider the abstraction of *pure chains* (and not of arbitrary graphs) in order to be able to keep the "topology" of pointer structures invariant in a more straightforward manner.

```
var $hd, tl, t$ :
( new($tl$); $hd := tl$;
  while ($true$) {
     new($tl\uparrow$);
     $tl := tl\uparrow$
  }
|| while ($true$) {
     if ($hd \neq tl$) {
        $t := hd$;
        $hd := hd\uparrow$;
        dispose($t$) }
   }
)
```

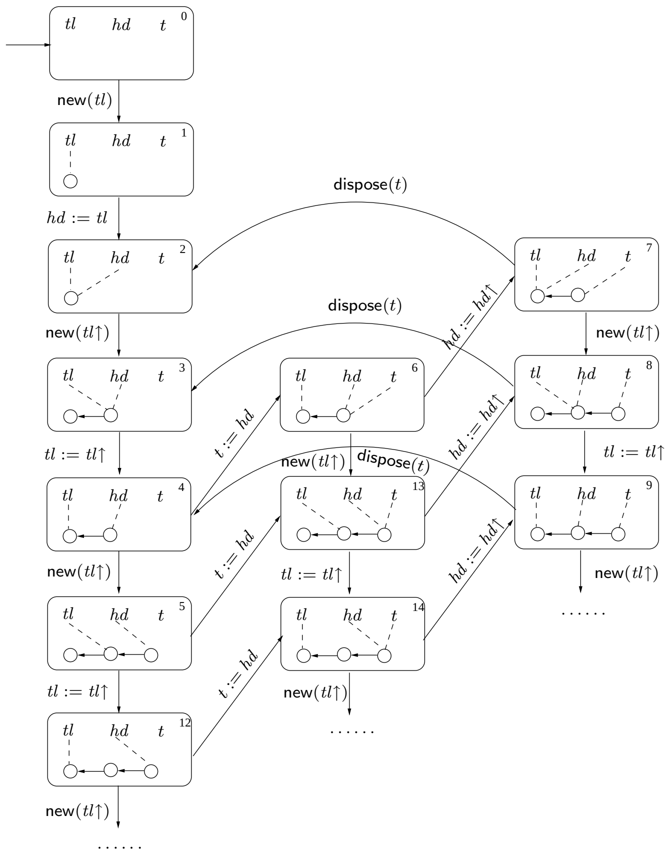


**Fig. 3.** Fragment of the automaton for the producer-consumer program

## 4.1 Abstracting pure chains

*Pure chains.* A sequence $e_1, \ldots, e_k$ of entities in a configuration is a chain (of length $k$) if $e_i \prec e_{i+1}$, for $0 < i < k$. The non-empty set $E$ of entities is a chain of length $|E| = k$ iff there exists a bijection $f : \{1, \ldots, k\} \rightarrow E$ such that $f(1), \ldots, f(k)$ is a chain; let $\mathit{first}(E) = f(1)$ and $\mathit{last}(E) = f(k)$. $E$ is a *pure chain* if $\mathit{indegree}_{\prec}(e) = 1$ for all $e \in f(2), f(3), \ldots, f(k)$ and $f$ is unique (which may fail to be the case if the chain is a cycle). Note that chains consisting of a single element are trivially pure.

*Abstracting pure chains.* An abstract entity may represent a pure chain of "concrete" entities. The concrete representation of abstract entity $e$ is indicated by its *cardinality* $\mathcal{C}(e) \in \mathbb{M} = \{1, \ldots, M\} \cup \{*\}$, for some fixed constant $M > 0$. Entity $e$ for which $\mathcal{C}(e) = m \leqslant M$ represents a chain of $m$ "concrete" entities; if $\mathcal{C}(e) = *$, $e$ represents a chain that is longer than $M$. (Such entities are similar to summary nodes [38], with the specific property that they always abstract from pure chains.) The special cardinality function $\mathbf{1}$ yields one for each entity. The precision of the abstraction is improved on increasing $M$ (because more configurations are distinguished); moreover, as we will discuss in the next section, to model check a given temporal property, $M$ has to be large enough to at least evaluate all atomic predicates in the property with certainty.

**Definition 4 (abstract configuration).** *An* abstract configuration *is a tuple* $c = \langle E, \prec, N, \mathcal{C}\rangle$ *such that* $\langle E, \prec, N\rangle$ *is a configuration and* $\mathcal{C} : E \to \mathbb{M}$ *is a mapping associating a cardinality to each* $e \in E$*, such that* $\mathcal{C}(e) = 1$ *if* $e \in N \cup PV$*.*

Evidently, each concrete configuration (cf. Def. 1) is an abstract configuration such that $\mathcal{C} = \mathbf{1}$.

Configurations representing pure chains at different abstraction levels are related by *morphisms*, defined as follows. Let *Cnf* denote the set of all configurations ranged over by $c$ and $c'$, and $\mathcal{C}(\{e_1, \ldots, e_n\}) = \mathcal{C}(e_1) \oplus \ldots \oplus \mathcal{C}(e_n)$ denote the number of concrete cells represented by $e_1$ through $e_n$, where $n \oplus m = n+m$ if $n+m \leqslant M$ and $*$ otherwise.

**Definition 5 (morphism).** *For* $c, c' \in$ *Cnf, a morphism from* $c$ *to* $c'$ *is a surjective function* $h : E \to E'$ *such that:*

1. *for all* $e \in E'$*,* $h^{-1}(e)$ *is a pure chain and* $\mathcal{C}'(e) = \mathcal{C}(h^{-1}(e))$
2. $e \prec' e' \;\Rightarrow\; last(h^{-1}(e)) \prec first(h^{-1}(e'))$
3. $e \prec e' \;\Rightarrow\; h(e) \preceq' h(e')$ *where* $\preceq'$ *denotes the reflexive closure of* $\prec'$
4. $h(e) \in N'$ *if and only if* $e \in N$*.*

According to the first condition only pure chains may be abstracted by a single entity, while keeping the cardinalities invariant. The second and third condition enforce the preservation of the pointer structure under $h$. The last condition asserts that the notion of freshness should be preserved. Intuitively speaking, by means of a morphism the abstract shape of the pointer dependencies represented by the two related configurations is maintained. The identity function *id* is a morphism and morphisms are closed under composition.

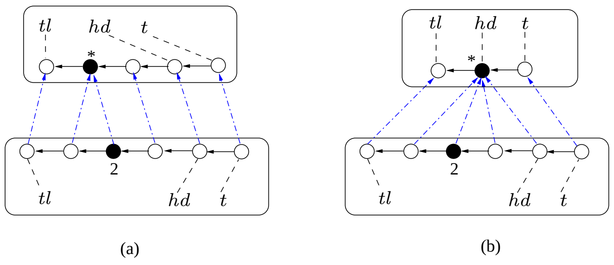


**Fig. 4.** Morphisms between configurations of the producer-consumer program at different abstraction levels

*Example 3.* Fig. 4(a) shows two configurations of the producer-consumer program with a shared list as buffer, at two levels of abstraction. It is assumed that $M$=2. The top configurations are abstractions of the bottom ones. Open circles denote concrete entities and filled circles denote abstract entities; their cardinality is indicated next to them. The morphism is indicated by dashed arrows. An alternative abstraction of the same configuration is depicted in Fig. 4(b). Although the indicated mapping is indeed a morphism in the sense of Def. 5, it is clear that this abstraction is too coarse as there is no way to distinguish between the cells pointed to by $hd$ and $hd\!\uparrow$, expressions that both occur in the producer-consumer program.

*Evolving pointer structures.* Morphisms relate configurations that model the pointer structure at distinct abstraction levels. They do not model the dynamic evolution of such linking structures. To reflect the execution of pointer-manipulating statements, such as either the creation or deletion of entities, or the change of pointers — the so-called "pointer swing" —by assignments (e.g., $x := x\!\uparrow\uparrow$), we use *reallocations.*

**Definition 6 (reallocation).** *For $c, c' \in \mathit{Cnf}$, $\lambda : (E^{\perp} \times E'^{\perp}) \to \mathbb{M}$ is a* reallocation *if:*

1. *(a) $\mathcal{C}(e) = \bigoplus_{e' \in E'^{\perp}} \lambda(e, e')$ and (b) $\mathcal{C}'(e') = \bigoplus_{e \in E^{\perp}} \lambda(e, e')$*
2. *(a) for all $e \in E$, $\{e' \mid \lambda(e, e') \neq 0\}$ is a chain, and*
   *(b) for all $e' \in E'$, $\{e \mid \lambda(e, e') \neq 0\}$ is a chain*
3. *for all $e \in E$, $|\{e' \mid \lambda(e, e') = *\}| \leqslant 1$*
4. *$\{e \mid \lambda(\perp, e) > 0\} = N'$*

*Let $\Lambda$ denote the set of reallocations, and $c \overset{\lambda}{\rightsquigarrow} c'$ denote that there exists a reallocation $\lambda$ between $c$ and $c'$.*

We explicitly use the undefinedness symbol $\perp$ to model birth (allocation) and death (deallocation) of entities: $\lambda(\perp, e) = n \neq 0$ denotes the birth of ($n$ instances of) $e$ whereas $\lambda(e, \perp) = n \neq 0$ denotes the death of ($n$ instances of) $e$. The conditions express that reallocation $\lambda$ redistributes cardinalities on $E$ to $E'$ such that (1a) the total cardinality sent by $\lambda$ from a source entity $e \in E$ equals $\mathcal{C}(e)$ and (1b) the total cardinality received by a target entity $e' \in E'$ equals $\mathcal{C}'(e')$; also, (2a) the entities that send at least one instance to a given target entity $e' \in E'$ form a chain in the source, and likewise, (2b) the entities that receive at least one entity from a given source $e \in E$ form a chain in the target. Moreover, (3) for each source entity $e$, at most one target entity $e'$ receives unboundedly many instances. Finally (4) expresses the correlation between the birth of entities and the freshness of those entities in the target. Note that, due to $\mathcal{C}'(e') = 1$ for $e' \in N'$ (see 4) and condition (1b) it follows that $\lambda(\perp, e') = 1$ and $\lambda(e, e') = 0$ for all $e \in E$.

In some cases we can derive a reallocation $\lambda_R$ between abstract configurations unambiguously from a binary relation $R$ between the sets of entities. In the lemma below we use $R(e)$ to denote $\{e' \mid (e, e') \in R\}$ and $R^{-1}(e')$ to denote $\{e \mid (e, e') \in R\}$.

**Lemma 1.** *Let $c, c'$ be two abstract configurations, and let $R \subseteq E \times E'$ be a binary relation. We call $R$* predictable *if it satisfies the following conditions for all $(e, e') \in R$:*

- *either* $|R(e)|{=}1$ *and* $\mathcal{C}(R^{-1}(e')){=}\mathcal{C}'(e')$ *or* $|R^{-1}(e')|{=}1$ *and* $\mathcal{C}'(R(e)){=}\mathcal{C}(e)$*;*
- $R(e)$ *is a* $\prec'$*-chain and* $R^{-1}(e')$ *is a* $\prec$*-chain;*
- $e'' \in R(e)$ *implies either* $e'' = e'$ *or* $\mathcal{C}'(e') = 1$ *or* $\mathcal{C}'(e'') = 1$*;*
- $e' \notin N'$*.*

*If* $R$ *is predictable, there is exactly one reallocation* $\lambda_R$ *between* $c$ *and* $c'$ *with:*

- $R = \{(e, e') \in E \times E' \mid \lambda(e, e') > 0\}$*;*
- $\lambda(\perp, e') > 0$ *if and only if* $R^{-1}(e') = \varnothing$*;*
- $\lambda(e, \perp) > 0$ *if and only if* $R(e) = \varnothing$*.*

Note that, in particular, any one-to-one relation between $c$ and $c'$ for which $\mathcal{C}(e) = \mathcal{C}'(e')$ if $(e, e') \in R$, is predictable. A special class are the so-called *functional reallocations*, which leave all cardinalities unchanged.

It is straightforward to check that, for any configuration $c$, the "identity" function that maps each pair $(e, e)$ for $e \in E_c$ onto $\mathcal{C}_c(e)$ is a reallocation.

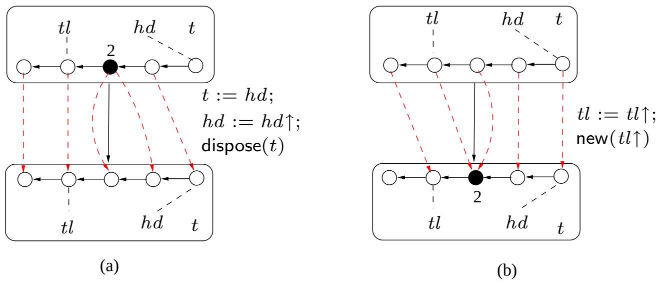


**Fig. 5.** Reallocations for evolution in the producer-consumer program

*Example 4.* Fig. 5(a) shows how an abstract configuration of the producer-consumer program (for $M{=}2$) evolves on performing two assignments and a disposal. The corresponding reallocation is depicted by dashed arrows between the configurations. Fig. 5(b) shows the reversed transition and its reallocation. Note that in (a) a cell is disposed (the one without outgoing dashed arrow), whereas in (b) a cell is created (the one without incoming dashed arrow).

The concept of reallocation can be considered as a generalization of the idea of identity change as, for instance, present in history-dependent automata [31]: besides the possible change of the abstract identity of concrete entities, it allows for the evolution of pointer structures. Reallocations allow "extraction" of concrete entities from abstract entities by a redistribution of cardinalities between entities. Extraction is analogous to *materialization* [38]. Reallocations ensure that entities that are born do not originate from any other entity. Moreover, entities that die can only be reallocated to $\perp$. This is the way in which birth and death of cells is modeled.

*Pointer automata.* In order to model the dynamic evolution of programs manipulating abstract representations of linked lists, we use (abstract) pointer automata. These are the same structures as before, except that each transition is now indexed with a reallocation, and states are equipped with abstract (rather than concrete) configurations.

**Definition 7 (abstract pointer automaton).** *An abstract pointer automaton $A = \langle Q, cf, \rightarrow, I, \mathcal{F}\rangle$ with $Q, cf, I$ and $\mathcal{F}$ as before (cf. Def 2), and transition relation $\rightarrow \subseteq Q \times \Lambda \times Q$, indexed by reallocations, such that:*

$$q \xrightarrow{\lambda} q' \quad \textit{implies that} \quad \lambda \textit{ is a reallocation from } cf(q) \textit{ to } cf(q')$$

Runs of abstract pointer automata are alternating sequences of states and reallocations, i.e., $q_0\lambda_0 q_1\lambda_1 q_2\cdots$ such that $q_i \xrightarrow{\lambda_i} q_{i+1}$ for all $i \geqslant 0$, $q_0 \in I$, and each accept set in $\mathcal{F}$ is visited infinitely often. Each run can be said to accept sequences of *concrete* configurations that are compatible with the reallocations, in a way to be defined below.

## 4.2 Symbolic semantics

Although the concrete semantics is rather simple and intuitive, it suffers from the problem that it easily results in an infinite state space. To circumvent this problem, we provide a semantics in terms of abstract pointer automata.

*Informal idea of the symbolic semantics.* As a start, we determine by means of a syntactic check through the program $p$ under consideration, the "longest" navigation expression that occurs in it and fix constant $L_p$ such that

$$L_p > \max\{n \mid v\uparrow^n \text{ occurs in program } p\}$$

Besides the formula-dependent constant $M$, the program-dependent constant $L_p$ can be used to tune the precision of the symbolic representation, i.e., by increasing $L_p$ the model becomes less abstract. Unbounded entities (i.e., those with cardinality $*$) will be exploited in the semantics to keep the model finite. The basic intuition of our symbolic semantics is that unbounded entities should always be preceded by a chain of at least $L_p$ concrete entities. Such states (or configurations) are called *safe*. This principle allows us to precisely determine the concrete entity that is referred to by any navigation expression in the program.[1] As assignments may yield unsafe configurations (due to program variables that are "shifted" too close to an unbounded entity), these statements require some special treatment (as we will see).

**Definition 8 (safe configuration).** *For fixed $L > 0$, configuration $c$ is $L$-*safe *if:*

$$\forall e \in PV. \forall e' : d(e, e') \leqslant L \;\Rightarrow\; \mathcal{C}(e') = 1$$

*where $d(e, e') = n$ if $e' = succ^n(e)$, and $d(e, e') = \bot$ if $e \not\prec^* e'$.*

[1] This is the sense in which the configuration is safe. The reader should not confuse the idea of safe configuration we use here with other concepts such as memory safety.

Here, $succ^0(e) = e$ and $succ^{n+1}(e) = succ(succ^n(e))$. That is to say, in an $L$-safe configuration, all entities within distance $L$ of a program variable are concrete.

*Example 5.* The upper configuration in Fig. 4(a) is 2-safe, since each program variable is at distance at least two from the abstract entity, but not 3-safe. (Recall that each program variable is an entity.) The upper configuration in Fig. 4(b) is not 1-safe. It follows by easy verification that all states in the concrete pointer automaton $[\![p]\!]^{\mathsf{conc}}$ for program $p$ are $L$-safe for any $L > 0$.

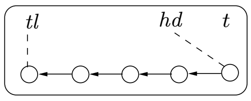
*Normal form.* For the symbolic semantics we consider configurations that, up to isomorphism, uniquely represent a set of "safe" states that may be related by morphisms. Such configurations are said to be in *normal form.* The notion of normal form is based on compactness:

**Definition 9 (compact configuration).** *For fixed $L > 0$, configuration $c$ is $L$-*compact *if for any entity $e$:*

$$indegree_{\prec}(e) > 1 \quad or \quad d(e', e) \leqslant L{+}1 \text{ for some } e' \in PV$$

Configuration $c$ is thus called *$L$-compact* if non-trivial pure chains appear within at most distance $L$+1 from some program variable. Cells that belong to a cycle and that are "entrances" to the cycle are compact, i.e., these cells will not be abstracted from.

*Example 6.* The upper configuration in Fig. 4(a) is 2-compact, as all entities are within distance at most three from a program variable. The following $L$-safe configuration (for any $L > 0$), on the other hand, is not $L$-compact for $L < 3$:



as two concrete cells are "too far" from a program variable, and thus need to be represented in a more compact way.

**Definition 10 (normal-form configuration).** *For fixed $L > 0$, configuration $c$ is in $L$-*normal form *whenever it is $L$-safe and $L$-compact.*

Given that the number of program variables is finite, and that we only consider cells that are reachable from program variables, it follows that:

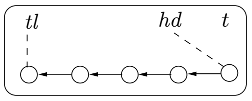
**Theorem 1.** *There are only finitely many $L$-normal form configurations.*

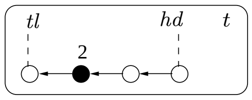
The fact that the normal form of an $L$-safe configuration is unique follows from the following:

**Theorem 2.** *For $c \in Cnf$: if $c$ is $L$-safe and reachable, then there is a unique $L$-normal $c' \in Cnf$ and a unique morphism between $c$ and $c'$.*

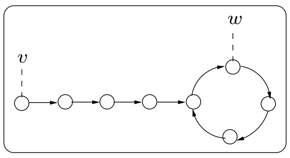
Intuitively speaking, configurations in $L$-normal form are the most compact representations of $L$-safe configurations. The normal form of $L$-safe configuration $c$ is denoted $\mathit{nf}(c)$, and $h_{\mathit{nf}}(c)$ denotes the corresponding unique morphism between $c$ and $\mathit{nf}(c)$.

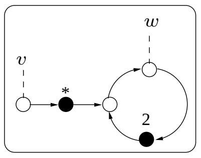
*Example 7.* Let $M$=2. In the following figure, the right hand configuration is the $L$-normal form of left hand one for $L$=2:





In the following figure, the right hand configuration is the $L$-normal form of left hand one for $L$=1:





For normal form configurations, we define the following relation between abstract reallocations and pairs of concrete configurations.

**Definition 11 (encoding).** *Let $c, c'$ be in $L$-normal form (for some $L$) and let $\lambda$ be a reallocation between $c$ and $c'$. $\lambda$ is said to* encode *a concrete pair of reallocations $c_1, c'_1$ if $c = \mathit{nf}(c_1)$ and $c' = \mathit{nf}(c'_1)$ with normal form morphisms $h_{\mathit{nf}}$ and $h'_{\mathit{nf}}$, respectively, and for all $e \in E$ and $e' \in E'$:*

$$\begin{aligned}\lambda(e, \bot) &= |h_{\mathit{nf}}^{-1}(e) \setminus E'_1| \\ \lambda(\bot, e') &= |{h'}_{\mathit{nf}}^{-1}(e') \setminus E_1| \\ \lambda(e, e') &= |h_{\mathit{nf}}^{-1}(e) \cap {h'}_{\mathit{nf}}^{-1}(e')|\end{aligned}$$

*(where the cardinalities on the right hand side are interpreted modulo $M$, i.e., they turn into $*$ if the cardinality exceeds $M$).*

For an abstract pointer automaton $A$ whose configurations are all in normal form, such as the ones we will use below to give a finite-state semantics to our language, using this notion of encoding we can define what it means for $A$ to *simulate* a concrete automaton (see Sect. 4.3), as well as the *language* of $A$. In particular, for the latter, we consider that a run $q_0\lambda_0q_1\lambda_1q_2\cdots$ of $A$ accepts a sequence of concrete configurations $c_0\, c_1\, c_2\, \cdots$ if each $\lambda_i$ encodes the pair $c_i, c_{i+1}$, and we define $\mathcal{L}(A)$ to be the set of configuration sequences accepted by some run of $A$.

*Safe expansions.* As argued above, performing an assignment to an $L$-safe state may lead to a state that is not $L$-safe due to program variables that are moved too close to an unbounded entity. This happens, for instance, when variable

$v$ is assigned an entity further down in the list originally pointed to by $v$. To overcome this difficulty, the semantics of assignment yields a *set* of possible successor configurations that are related to each other in some sense. This is the main source of nondeterminism (i.e., over-approximation). These configurations, together with the morphisms that relate them to the configuration $c$ in which the assignment is executed, are the *safe expansions* of $c$.

**Definition 12 (safe expansion).** *For fixed $L > 0$ and configuration $c$, $\Uparrow c$ is the set of pairs $(c', h)$ such that $c'$ is $L$-safe and $h$ is a morphism from $c'$ to $c$ with shrink factor at most $L$.*

The shrink factor of morphism $h$ is defined as $\max\{|h^{-1}(e)| - 1 \mid e \in E'\}$. It is important to note that $\Uparrow c$ is finite (up to isomorphism).

*Operational semantics.* With the use of safe expansions we are now in a position to define the symbolic semantics of our programming language. A key observation is that the definitions of $add(c, \ell)$, $cancel(c, \alpha)$ and $modify(c, \ell, \alpha)$ can also be applied if $c$ is abstract, provided it is $L$-safe for some $L$ no smaller than the number of consecutive dereferencing operations in $\ell$ and $\alpha$ — so that $[\![\ell]\!]^{\mathsf{loc}}$, $[\![\ell]\!]^{\mathsf{exp}}$ and $[\![\alpha]\!]^{\mathsf{exp}}$ all point to a uniquely determined, concrete entity. For that reason we can use the relation $\longrightarrow$ as derived according to Tables 1 and 2 over abstract configurations, as long as we ensure $L$-safety for sufficiently large $L$. Furthermore, if we derive a transition $s, c \longrightarrow s', c'$ using these rules, then the identity relation $\{(e, e) \mid e \in E \cap E'\}$ is predictable in the sense of Lemma 1.

**Definition 13.** *The symbolic semantics of the program*

$$p = \mathsf{decl}\ v_1, \ldots, v_n : (s_1 \parallel \cdots \parallel s_k)$$

*is the (abstract) pointer automaton $[\![p]\!]^{\mathsf{symb}} = \langle Q, cf, \rightarrow, I, \mathcal{F} \rangle$ where $Q, cf, I$ and $\mathcal{F}$ are defined as for the concrete semantics (see Def. 3) and $\rightarrow \subseteq Q \times \Lambda \times Q$ is the smallest relation satisfying:*

$$\frac{s, c \longrightarrow s', c' \ \wedge\ (c'', h) \in \Uparrow c'}{s, c \longrightarrow_{\lambda_R} s', nf(c'')} \quad \textit{where} \quad R = h_{nf} \circ h^{-1} \circ id_{E \cap E'}\ .$$

Let us explain this rule. The idea is that, by construction, all abstract configurations generated by the semantics are in $L$-normal form, implying that they are $L$-safe for sufficiently large $L$, so that we can indeed apply the concrete operational semantics (as discussed above). The abstract configuration thus derived, however, is no longer in $L$-normal form; therefore we take all safe expansions (introducing non-determinism) and normalize them. These steps (derivation–expansion–normalization) are accompanied by, respectively, a one-to-one identity relation or partial function ($id_{E \cap E'}$), an inverse morphism ($h^{-1}$) and a morphism ($h_{nf}$). By the definition of safe expansion it follows that $h(e) = h(e')$ for distinct $e, e'$ implies (i) either $e$ or $e'$ has cardinality 1, and (ii) $h_{nf}(e) \neq h_{nf}(e')$. From this and the fact that both $h$ and $h_{nf}$ are morphisms, it can be deduced that $h_{nf} \circ h^{-1} \circ id_{E \cap E'}$ is predictable in the sense of Lemma 1, and hence $\lambda_R$ is well-defined.

It is noteworthy that the safe expansion step is only really necessary if the original, concrete transition has been caused by an underlying *modify*() operation (i.e., is the result of an assignment): the *add*() and *cancel*() operations cannot result in unsafe configurations, and hence no expansion is necessary afterwards. It is, therefore, only assignment statements that cause non-determinism in the abstract semantics.

*Example 8.* Consider the producer-consumer program where the buffer is modeled as a shared (unbounded) list:

```
var $hd, tl, t$ :
(  $\mathsf{new}(tl); hd := tl;$ while $(true)$ {$\mathsf{new}(tl\uparrow); tl := tl\uparrow$ }                // producer
 || while $(true)$ if $(hd \neq tl)$ {$\langle t := hd; hd := hd\uparrow\rangle; \mathsf{dispose}(t)$ } // consumer
)
```

An initial fragment of the (abstract) pointer automaton for this program has already been provided in Example 2. For $L$=2 and $M$=1, Fig. 6 illustrates the part of the abstract pointer automaton in which abstraction plays a role. (The entire pointer automaton has 30 states.) With respect to the version given before, we have introduced atomicity in the consumer, which now atomically takes an item from the list and shifts the *hd* of the buffer. To avoid cluttering up the figure, the reallocations and the program statements are omitted, as are the accept states. The same applies to the intermediate states of the atomic regions.

Note that a collector cell is introduced as soon as two concrete cells can be "summarized" without violating the 2-safeness constraint. This happens, e.g., when performing the assignment $tl := tl\uparrow$ in configuration 22. A case of nondeterminism that arises from considering safe expansions for assignments are the two transitions, both labeled with the statement $\langle t := hd; hd := hd\uparrow\rangle$, emanating from configuration 28. As the source configuration contains a collector cell, this cell represents a list of two or more cells. Both possibilities are considered: for a list of exactly two elements, configuration 22 results; the other case corresponds to configuration 26.

## 4.3 Properties of the semantics

The symbolic semantics gives us an analysis that by itself already yields some useful information on the program, such as the possibility of memory violation (e.g., if one of the parallel components of the program in a reachable state equals $\mathsf{error}$). This analysis has two important properties: it is sound since it represents an over-approximation of the concrete semantics; and it is finite, and therefore computable.

In more detail, the concrete and abstract pointer automata generated by the concrete and symbolic semantics of a given program, respectively, are related by a forward simulation defined using the notion of encoding in Def. 11. Let $q$ be

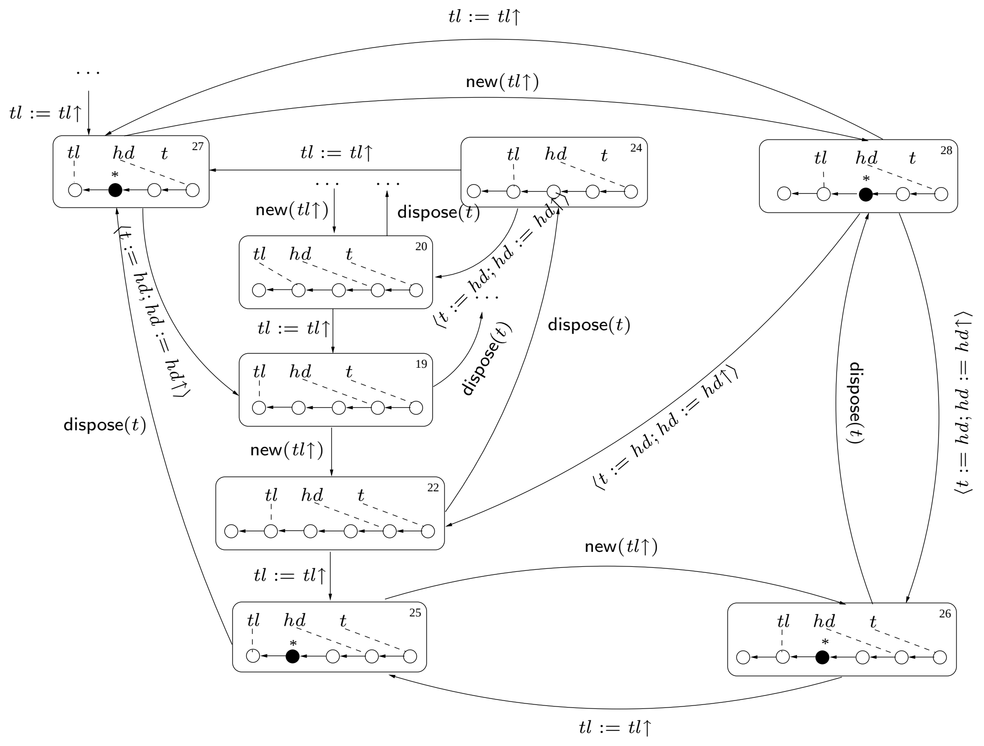


**Fig. 6.** Abstract pointer automaton for producer-consumer with shared list.

a state in the concrete automaton. The abstract state $q_{abs}$ is said to simulate $q$ whenever

(i) $cf(q_{abs}) = nf(cf(q))$, and
(ii) for every transition $q \longrightarrow q'$, there exists a reallocation $\lambda$ that encodes the pair $cf(q)$, $cf(q')$, such that $q_{abs} \longrightarrow_\lambda q'_{abs}$ and $q'_{abs}$ simulates $q'$.

Pointer automaton $A^{\mathsf{symb}}$ simulates $A^{\mathsf{conc}}$, denoted $A^{\mathsf{conc}} \sqsubseteq A^{\mathsf{symb}}$, whenever there exists a forward simulation relation satisfying (i) and (ii) for all pairs $(c, c_{abs})$ in the relation, such that initial states and accept states correspond. The following is then straightforward to prove:

**Theorem 3.** *If $A^{\mathsf{conc}} \sqsubseteq A^{\mathsf{symb}}$, then $\mathcal{L}(A^{\mathsf{conc}}) \subseteq \mathcal{L}(A^{\mathsf{symb}})$.*

The relation between the concrete and symbolic semantics can be expressed in terms of this notion of forward simulation (for details see [18]):

**Theorem 4.** *For any program $p$: $[\![p]\!]^{\mathsf{conc}} \sqsubseteq [\![p]\!]^{\mathsf{symb}}$.*

Moreover, we have the following crucial property of the symbolic semantics:

**Theorem 5.** *For any program $p$: $[\![p]\!]^{\mathsf{symb}}$ is finite state.*

More specifically, the number of states of the symbolic semantics is bounded by $k \cdot 2^K \cdot \sum_{n=0}^{K}(n+1)^n \cdot \sum_{n=0}^{K}(M+1)^n$ where $k$ is a constant dependent on the length of the longest sequential component and $K$ is an upper-bound on the number of entities in each state. Note that $K$ is bounded since the number of program variables is finite, and there cannot be an infinite-length chain in a state, due to normal form.

## 5 Pointer Logic

To express properties of concurrent pointer programs, we use a first-order extension of linear temporal logic [35]. The logic allows to express properties over sequences of configurations. The intention is that these sequences are generated by the pointer automata.

### 5.1 Syntax of the pointer logic

In the logic, heap cells (i.e., entities) are referred to by *logical* variables, taken from a countable set *LV*, ranged over by $x, y, z$, such that $LV \cap PV = \varnothing$. The connection between logical variables and cells is established by a partial valuation, meaning that logical variables, like program variables, may be undefined. Logical variables are a special case of pointer expressions, i.e., expressions that refer to heap cells. The syntax of pointer expressions is defined as before by the grammar:

$$\alpha \ ::= \ nil \ \Big| \ x \ \Big| \ \alpha\uparrow$$

where *nil* denotes the special entity in *Ent*, $x$ denotes the cell assigned by the current valuation (which may be *nil* or undefined), and $\alpha\uparrow$ denotes the entity referred to by (the entity denoted by) $\alpha$ (if any). Thus, $x\uparrow^n$ denotes the $(n{+}1)$-st cell in the list referred to by $x$.

The syntax of the logic *Navigation Temporal Logic* (NTL, for short) is defined by the grammar:

$$\Phi \ := \ \alpha = \alpha \ \Big| \ \alpha \leadsto \alpha \ \Big| \ \mathsf{undef}\,\alpha \ \Big| \ \mathsf{new}\,\alpha \ \Big| \ \exists x.\, \Phi \ \Big| \ \Phi \wedge \Phi \ \Big| \ \neg\Phi \ \Big| \ \bigcirc\Phi \ \Big| \ \Phi \,\mathsf{U}\, \Phi$$

The proposition $\alpha = \beta$ states that $\alpha$ and $\beta$ are aliases. Here, equality is strict. Proposition $x\uparrow^2 = y\uparrow^3$, for example, denotes that the third cell in $x$'s list is also the fourth cell in $y$'s list. The proposition $\alpha \leadsto \beta$ expresses that (the cell denoted by) $\beta$ is reachable from (the cell denoted by) $\alpha$ via the pointer structure. Thus, $x \leadsto y\uparrow^3$ expresses that in the current state the fourth cell in $y$'s list can be reached by following the pointer structure from the cell denoted by $x$. Proposition $\mathsf{undef}\,\alpha$ states that $\alpha$ is dangling (i.e., undefined), and $\mathsf{new}\,\alpha$ asserts that the cell referred to by $\alpha$ is fresh. The existential quantification $\exists x.\Phi$ is valid if an appropriate cell for $x$ can be found such that $\Phi$ holds. The boolean connectives, and the linear temporal connectives $\bigcirc$ (next) and $\mathsf{U}$ (until) have the usual interpretation. We denote $\alpha \neq \beta$ for $\neg\,(\alpha = \beta)$, $\alpha \not\leadsto \beta$ for $\neg\,(\alpha \leadsto \beta)$,

$\mathsf{alive}\,\alpha$ for $\neg(\mathsf{undef}\,\alpha)$, and $\forall x.\,\Phi$ for $\neg\,(\exists x.\,\neg\Phi)$. The other boolean connectives (such as disjunction, implication and equivalence) and the temporal operators $\Diamond$ (eventually) and $\Box$ (always) are obtained in the standard way.

Note that *NTL* is in fact a *quantified modal logic* (see, e.g., [3, 21]) as quantification and temporal operators can be mixed arbitrarily. In particular, temporal operators can be used inside quantification.

*Example 9.* We illustrate the expressiveness of the logic NTL by a number of example properties that are frequently encountered for pointer manipulating programs.

- The third cell in $x$'s list and the head of $y$'s list eventually become aliases:

$$\Diamond(x\!\uparrow\uparrow = y)$$

.
- $x\!\uparrow$ will never be dangling:

$$\Box(\mathsf{alive}\,x\!\uparrow)$$

.
- Eventually, $v$ will be part of a non-empty cycle:

$$\Diamond(\exists x.\,x \neq v \;\wedge\; x \rightsquigarrow v \;\wedge\; v \rightsquigarrow x)$$

- Every cell reachable from $v$ will be eventually disposed:

$$\forall x.\,(v \rightsquigarrow x \Rightarrow \Diamond\mathsf{undef}\,x)$$

- Whenever $y$ is a cell in $x$'s list, $y$ and $x$ can only become disconnected when $y$ is disposed:

$$(\forall x.\,\forall y.\,x \rightsquigarrow y \Rightarrow (\Box\mathsf{alive}\,y \;\vee\; (x \rightsquigarrow y)\,\mathsf{U}\,\mathsf{undef}\,y))$$

- An unbounded number of cells will be created:

$$\Box\Diamond(\exists x.\,\mathsf{new}\,x)$$

- Cells are disposed in the order of creation:

$$\Box\,(\forall x.\,\mathsf{new}\,x \;\Rightarrow\; \Box\,(\forall y.\,\mathsf{new}\,y \;\Rightarrow\; (\mathsf{alive}\,y\,\mathsf{U}\,\mathsf{undef}\,x)))$$

  This can be understood as follows: any entity $x$ that is fresh in the current state will be dead before (or at the same time as) any younger entity $y$ (fresh in some later state) dies.

*Program variables.* To enable reasoning over program variables (rather than just logical ones), we introduce for each relevant program variable $v$ a logical variable $x_v$, which always evaluates to the entity $v \in E$. We then use $v$ in the logic as syntactic sugar for $x_v\!\uparrow$, so that it has the expected value. Furthermore, when we write $\exists\, x.\, \Phi$ we really mean $\exists\, x.\, (x \neq x_{v_1} \wedge \ldots \wedge x \neq x_{v_n}) \ \Rightarrow\ \Phi$, where $\{v_1, \ldots, v_n\}$ is the set of program variables occurring in the program.

*Example 10.* Consider the list-reversal program (cf. Section 2) that intends to reverse the list initially pointed to by variable $v$. Properties of interest of this program include, for instance:

– $v$ and $w$ always point to distinct lists (heap non-interference):

$$\Box(\forall x.\, v \rightsquigarrow x \ \Rightarrow\ w \not\rightsquigarrow x)$$

– $v$'s list will be (and remains to be) reversed and the resulting list will be given to $w$ [2]:

$$\forall x.\, \forall y.\ ((v \rightsquigarrow x \ \wedge\ x\!\uparrow = y) \ \Rightarrow\ \Diamond\Box(y\!\uparrow = x \wedge w \rightsquigarrow y))$$

Note that the previous formula expresses the precise specification of the list reversal program. In particular, it implies that the reversed list contains precisely the same elements of the original list and that their pointers are properly reversed. This property is not usually verifiable by shape analyses that do not keep track of the evolution of entities during the program computation.

– none of the cells in $v$'s list will ever be deleted:

$$\forall x.\, (v \rightsquigarrow x \ \Rightarrow\ \Box \mathsf{alive}\, x)$$

Properties for the producer-consumer program with a shared list are:

– every element in the buffer is eventually consumed:

$$\Box(hd \neq tl \ \Rightarrow\ \exists x.\, (x = hd \ \wedge\ \Diamond \mathsf{undef}\, x))$$

(Note that this is *not* the same as $\Box(hd \neq tl \ \Rightarrow\ \Diamond \mathsf{undef}\ hd)$; in the former property, $x$ is frozen to the value of $hd$ in the state where it is bound, and so the property expresses that *that* particular entity dies; the latter expresses that $hd$ itself may become undefined.)

– the tail is never deleted nor disconnected from the head:

$$\Box(\mathsf{alive}\, tl \ \wedge\ hd \rightsquigarrow tl)$$

Taking into account the semantics of the logic, to be defined below, from Fig. 6 it can be observed that both formulae are valid in the abstract pointer automaton that models the producer-consumer program. Using Theorem 4 and Corollary 1, we conclude that the original program (as represented in the concrete semantics) also exhibits these properties. The same applies to the ordering property that requires elements to be consumed in the order of production.

[2] If one is interested in only checking whether $v$'s list is reversed at the end of the program, program locations can be added and referred to in the standard way.

## 5.2 Semantics of the pointer logic

Logical formulae are interpreted over infinite sequences of configurations. We need a function $\theta$ that is a partial valuation of the logical variables, i.e., $\theta(x)$ is either undefined or equals some cell, which is then the value of $x$ — as we shall see, this is always an entity in the initial configuration of the sequence under consideration.

The semantics of navigation expression $\alpha$ is given by:

$$
\begin{aligned}
[\![ nil ]\!]_{\prec,\theta} &= nil \\
[\![ x ]\!]_{\prec,\theta} &= \theta(x) \\
[\![ \alpha\uparrow ]\!]_{\prec,\theta} &= succ\,([\![\alpha]\!]_{\prec,\theta})
\end{aligned}
$$

Let $\sigma = c_0\,c_1\,c_2\cdots$ be a sequence of concrete configurations. The semantics of *NTL*-formulae is defined by the satisfaction relation $\sigma,\theta \models \Phi$, defined as follows:

$$
\begin{array}{lll}
\sigma,\theta \models \alpha = \beta & \text{iff} & [\![\alpha]\!]_{\prec_0,\theta} = [\![\beta]\!]_{\prec_0,\theta} \\
\sigma,\theta \models \alpha \rightsquigarrow \beta & \text{iff} & \exists\, k \geqslant 0.\, [\![\alpha\uparrow^k]\!]_{\prec_0,\theta} = [\![\beta]\!]_{\prec_0,\theta} \\
\sigma,\theta \models \mathsf{undef}\,\alpha & \text{iff} & [\![\alpha]\!]_{\prec_0,\theta} = \bot \\
\sigma,\theta \models \mathsf{new}\,\alpha & \text{iff} & [\![\alpha]\!]_{\prec_0,\theta} \in N_0 \\
\sigma,\theta \models \exists x.\,\Phi & \text{iff} & \exists e \in E_0 : \sigma,\theta\{e/x\} \models \Phi \\
\sigma,\theta \models \Phi \wedge \Psi & \text{iff} & \sigma,\theta \models \Phi \text{ and } \sigma,\theta \models \Psi \\
\sigma,\theta \models \neg\Phi & \text{iff} & \sigma,\theta \not\models \Phi \\
\sigma,\theta \models \bigcirc\,\Phi & \text{iff} & \sigma^1, \tilde{\theta}_1 \models \Phi \\
\sigma,\theta \models \Phi\,\mathsf{U}\,\Psi & \text{iff} & \exists i.\,(\sigma^i, \tilde{\theta}_i \models \Psi \text{ and } \forall j < i.\, \sigma^j, \tilde{\theta}_j \models \Phi).
\end{array}
$$

Here, $\tilde{\theta}_i$ is defined by $\tilde{\theta}_0 = \theta$ and $\tilde{\theta}_{i+1} = \tilde{\theta}_i(x) \cap (LV \times E_{i+1})$; i.e., as soon as an entity is deallocated in the sequence (at some step $j \leqslant i$), it can no longer occur as an image in $\theta_i$. The substitution $\theta\{e/x\}$ is defined as usual, i.e., $\theta\{e/x\}(x) = e$ and $\theta\{e/x\}(y) = \theta(y)$ for $y \neq x$. $\sigma^i$ denotes the suffix of $\sigma$ that is obtained by erasing the first $i$ items from $\sigma$. Note that the proposition $\alpha \rightsquigarrow \beta$ is satisfied if $[\![\beta]\!] = \bot$ and $[\![\alpha]\!]$ can reach some cell with an undefined outgoing reference.

## 5.3 Properties

For pointer automaton $A$ and *NTL*-formula $\Phi$, $A \models \Phi$ holds whenever for *all* allocation sequences $\sigma$ of configurations in $\mathcal{L}(A)$ we have $\sigma,\theta \models \Phi$. The following is then an immediate consequence of Theorem 3.

**Corollary 1.** *For any NTL-formula $\Phi$ and pointer automata $A$ and $A'$:*

$$A \sqsubseteq A' \;\Rightarrow\; (A' \models \Phi \;\Rightarrow\; A \models \Phi)$$

In particular, as for any program $p$ we have that $[\![p]\!]^{\mathsf{conc}} \sqsubseteq [\![p]\!]^{\mathsf{symb}}$ (Theorem 4), it follows that any *NTL*-formula $\Phi$ that is valid for (the finite-state!) $[\![p]\!]^{\mathsf{symb}}$, it holds that $\Phi$ is valid in the (possibly infinite-state) program $p$. As this applies to all *NTL*-formulae, this includes safety and liveness properties.

## 6 Model Checking Pointer Logic

For the setup proposed in this paper we have developed a model checking algorithm, using tableau graphs as in [27] to establish whether or not a formula $\Phi$ is valid on a given (finite) abstract pointer automaton $A$. The algorithm is described in detail in [19]; here we give a brief summary.

*The parameters $M$ and $L$.* In the previous section, we have stressed that the precision of automaton $A$ is ruled by two parameters: $L$, which controls the distance between entities before they are collected into unbounded entities, and $M$, which controls the information we have about unbounded entities. As described in Sect. 4, $L$ is used in the generation of models from programs; it is no longer of importance in the model checking stage (where we supposed to have the model already). $M$, on the other hand, is a formula-dependent constant that must exceed $\sum_{x \in \Phi} \max\{i \mid x{\uparrow}^i \text{ occurs in } \Phi\}$ *for the formula $\Phi$ that we want to check on the model $A$.* This may mean that the $A$ at hand is not (yet) suitable for checking a given formula $\Phi$, namely if $M$ for that model does not meet this lower bound. In that case we have to *stretch* the model.

*Example 11.* Consider, for instance, the model depicted in Fig. 6. If we want to check whether the buffer may have size 5, this can be expressed by the formula $\Diamond(hd{\uparrow}^5 \leadsto tl)$; but in states where entities of the buffer have been collected into an unbounded entity (states 25–29 in the figure), it is not clear whether $hd{\uparrow}^5$ is pointing to (some entity within) that unbounded entity, or to some entity following it, in particular to $tl$.

To overcome this problem, we can stretch a given model without loss of information (but with loss of compactness, and hence increase of complexity of the model checking). Let, $\mathcal{C}(A)$ be the maximal concrete cardinality of some entity in $A$. In [19], the operation $A \Uparrow \widehat{M}$ is defined, which stretches $A$ such that $\mathcal{C}(A \Uparrow \widehat{M})$ is $\widehat{M}$. The resulting pointer automaton copies each state in $A$ that contains an unbounded entity $e$, such that for each materialization of $e$ from $M, M{+}1, \ldots, \widehat{M}$ and $*$ a state exists. We then have the following result:

**Theorem 6.** *For all abstract pointer automata $A$ such that $\mathcal{C}(A) < \widehat{M}$: $\mathcal{L}(A) = \mathcal{L}(A \Uparrow \widehat{M})$.*

The automaton $A \Uparrow \widehat{M}$ is a factor $n^{\widehat{M}-M}$ times as large as $A$, where $n$ is the maximum number of unbounded entities in the abstract configurations of $A$.

*The tableau graph.* The next step is to construct a *tableau graph* $G_A(\Phi)$ for $\Phi$ from a given pointer automaton $A$, assuming that stretching has been done, so $M$ satisfies the given lower bound for $\Phi$. $G_A(\Phi)$ enriches $A$, for each of its states $q$, with information about the collections of formulae relevant to the validity of $\Phi$ that possibly hold in $q$. These "relevant formulae" are essentially subformulae of $\Phi$ and their negations; they are collected into the so-called *closure* of $\Phi$. For instance, the closure of the formula $tl\,\mathsf{alive} \Rightarrow \Box(tl\,\mathsf{alive})$ which expands to $\neg\Psi \vee \neg(true\,\mathsf{U}\,\neg\Psi)$ with $\Psi = tl\,\mathsf{alive}$, is the set

$$\begin{array}{cccccc} true & \Psi & true\,\mathsf{U}\,\neg\Psi & \bigcirc(true\,\mathsf{U}\,\neg\Psi) & \bigcirc\neg(true\,\mathsf{U}\,\neg\Psi) & \Phi \\ \neg true & \neg\Psi & \neg(true\,\mathsf{U}\,\neg\Psi) & \neg\bigcirc(true\,\mathsf{U}\,\neg\Psi) & \neg\bigcirc\neg(true\,\mathsf{U}\,\neg\Psi) & \neg\Phi \end{array} .$$

In general, the size of the closure is linear in the size of the formula (as in [27]). The states of $G_A(\Phi)$ are called *atoms* $(q, D)$ where $q$ is a state of $A$ and $D$ a consistent and complete set of valuations of formulae from the closure of $\Phi$ on (the entities of) $q$. Consistency and completeness approximately mean that, for instance, if $\Psi_1$ is in the closure then exactly one of $\Psi_1$ and $\neg\Psi_1$ is "included in" $D$ (i.e., $D$ contains a valuation for it), and if $\Psi_1 \vee \Psi_2$ is in the closure then it is "in" $D$ iff $\Psi_1$ or $\Psi_2$ is "in" $D$, etc. For the precise definition we refer to [19]. For any $q$, the number of atoms on $q$ is exponential in the size of the closure and in the number of entities in $q$.

A transition from $(q, D)$ to $(q', D')$ exists in the tableau graph $G_A(\Phi)$ if $q \longrightarrow_\lambda q'$ in $A$ and, moreover, to the valuation of each sub-formula $\bigcirc\Psi$ in $D$ there exists a corresponding valuation of $\Psi$ in $D'$ — where the correspondence is defined modulo the reallocation $\lambda$.

A *fulfilling path* in $G_A(\Phi)$ is then an infinite sequence of transitions, starting from an initial state, that also satisfies all the "until" sub-formulae $\Psi_1\,\mathsf{U}\,\Psi_2$ in the atoms, in the sense that if a valuation of $\Psi_1\,\mathsf{U}\,\Psi_2$ is in a given atom in the sequence, then a corresponding valuation of $\Psi_2$ occurs in a later atom — where correspondence is the same notion as above, but now modulo a sequence of reallocations. We have the following result:

**Proposition 1.** *$A \models \Phi$ iff there does not exist a fulfilling path in $G_A(\neg\Phi)$.*

Hence the validity of the formula $\Phi$ is related to the existence of a fulfilling path in the graph $G_A(\neg\Phi)$. To decide this, we seek for the existence of a *self-fulfilling strongly connected sub-component* (SCS) of the tableau graph that is *reachable from an initial state through some prefix trace*. This gives a necessary criterion for the existence of a fulfilling path. In particular, if we use $Inf(\pi)$ to denote the set of atoms that occur infinitely often in an (arbitrary) infinite path $\pi$ in $G_A(\Phi)$, then we have:

**Proposition 2.** *$Inf(\pi)$ is not a self-fulfilling SCS $\;\Rightarrow\;$ $\pi$ is not a fulfilling path.*

Since the number of SCSs of any finite tableau graph is finite, and the property of self-fulfillment is decidable, this gives rise to a mechanical procedure for verifying the validity of formulae. This is formulated in the following theorem:

**Theorem 7.** *For any finite abstract pointer automaton $A$, it is possible to verify mechanically whether $A \models \Phi$.*

This, combined with Th. 4, implies that, for any concrete automaton $A^{\mathsf{conc}}$ of which $A$ is an abstraction, it is also possible to verify mechanically whether $A^{\mathsf{conc}} \models \Phi$. Note that although this theorem leaves the possibility of *false negatives* (as usual in model checking in the presence of abstraction), it does not produces false positives. This applies to both safety and liveness properties. Having false negatives means that if the algorithm fails to show $A \models \Phi$ then it cannot be concluded that $\Phi$ is *not* satisfiable (by some run of $A$). However, since such a failure is always accompanied by a "prospective" fulfilling path of $\neg\Phi$, further analysis or testing may be used to come to a more precise conclusion.

The algorithm is summarized in Table 3.

```
procedure valid($A, \Phi$)
begin
  construct $G_A(\neg\Phi)$;
  construct the set $\Pi$ of reachable self-fulfilling SCS
    satisfying the accept condition on $\mathcal{F}_A$;
  if   $\Pi = \varnothing$
  then return: "$\Phi$ is valid in $A$";
  else return $G' \in \Pi$ with its prefix as a (possible) counterexample;
  fi
end
```

**Table 3.** Procedure for validity of $\Phi$ in $A$.

## 7 Concluding Remarks

In this paper, we have introduced a sound analysis of concurrent programs manipulating heap-allocated linked lists. The analysis is based on an automaton model where states are equipped with abstract heap representations and transitions with mappings that allow to model the evolution of heap during the program computation. Moreover, the analysis is parametric in two constants. This latter feature reduces the process of abstraction-refinement to simply increasing/decreasing these parameters.

Furthermore, we define a temporal logic called *NTL* with pointer assertions as well as predicates referring to the birth or death of memory cells. Although *NTL* is essentially a first-order logic, it contains two second-order features: the reachability predicate $\alpha \rightsquigarrow \beta$ (which computes the transitive closure of pointers), and the freshness predicate new $\alpha$ (which expresses membership of the *set* of fresh entities).

For *NTL*, we introduce a sound (but not complete) model-checking algorithm to verify formulae against our automata models. Thus, safety and liveness properties of heap mutating programs can be verified. We like to mention that for the (much) simpler framework in which pointers are ignored, it is possible to check dynamic properties such as the creation and disposal of heap cells in a sound *and complete* manner, as described in [20].

## References

1. S. Bardin, A. Finkel, and D. Nowak. Towards symbolic verification of programs handling pointers. In: *AVIS 2004*.

2. A. Barr. *Find the Bug in this Java Program.* Addison-Wesley, 2005.
3. D. Basin, S. Matthews and L. Vigano. Labelled modal logics: quantifiers. *J. of Logic, Language and Information*, **7**(3);237–263, 1998.
4. J. Berdine, C. Calcagno, P.W. O'Hearn. A decidable fragment of separation logic. In: *FSTTCS*, LNCS 3328, pp. 97-109, 2004.
5. J. Berdine, C. Calcagno, P.W. O'Hearn. Symbolic execution with separation logic. *APLAS*, LNCS 3780, pp. 52-68, 2005.
6. J. Bergstra, A. Ponse and S.A. Smolka (editors). *Handbook of Process Algebra.* Elsevier, 2001.
7. A. Bouajjani, P. Habermehl, P. Moro and T. Vojnar. Verifying programs with dynamic 1-selector-linked list structures in regular model checking. In: *TACAS*, LNCS 3440, pp. 13–29, 2005.
8. M. Bozga, R. Iosif, and Y. Lakhnech. Storeless semantics and alias logic. In: *PEPM*, pp. 55–65. ACM Press, 2003.
9. M. Bozga, R. Iosif and Y. Lakhnech. On logics of aliasing. In: *SAS*, LNCS 3148, pp. 344-360, 2004.
10. R. Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence* **6**: 23–50, 1971.
11. L. Cardelli, P. Gardner, and G. Ghelli. A spatial logic for querying graphs. In: *ICALP*, LNCS 2380, pp. 597–610. Springer, 2002.
12. L. Cardelli and A.D. Gordon. Anytime, anywhere: modal logics for mobile ambients. In: *POPL*, pp. 365–377. ACM Press, 2000.
13. D.R. Chase, M. Wegman and F. Zadeck. Analysis of pointers and structures. In: *PLDI*, pp. 296–310. ACM Press, 1990.
14. S. Chong and R. Rugina. Static analysis of accessed regions in recursive data structures. In: *SAS*, LNCS 2694, pp. 463–482, 2003.
15. S.A. Cook and D. Oppen. An assertion language for data structures. In: *POPL*, pp. 160–166. ACM Press, 1975.
16. A. Deutsch. Interprocedural may-alias analysis for pointers: beyond $k$-limiting. In: *PLDI*, pp. 230–241. ACM Press, 1994.
17. D. Distefano. A parametric model for the analysis of mobile ambients. In: *APLAS*, LNCS 3780, pp. 401–417, 2005.
18. D. Distefano, J.-P. Katoen, and A. Rensink. Who is pointing when to whom? – On the automated verification of linked list structures In: *FSTTCS*, LNCS 3328, pp. 250–262, 2004.
19. D. Distefano, A. Rensink and J.-P. Katoen. Who is pointing when to whom? – On the automated verification of linked list structures CTIT Tech. Rep. 03-12, 2003.
20. D. Distefano, A. Rensink, and J.-P. Katoen. Model checking birth and death. In: *TCS*, pp. 435–447. Kluwer, 2002.
21. M. Fitting. On quantified modal logic. *Fundamenta Informatica*, **39**(1):5–121, 1999.
22. P. Fradet, R. Gaugne, and D. Le Métayer. Static detection of pointer errors: an axiomatisation and a checking algorithm. In: *ESOP*, pp. 125–140, LNCS 1058, 1996.
23. R.J. van Glabbeek. The linear time-branching time spectrum I. In [6], Chapter 1, pp. 3–101, 2001.
24. S. Ishtiaq and P.W. O'Hearn. BI as an assertion language for mutable data structures. In: *POPL*, pp. 14–26, ACM Press, 2001.
25. J. Jensen, M. Jørgensen, M. Schwartzbach and N. Klarlund. Automatic verification of pointer programs using monadic second-order logic. In: *PLDI*, pp. 226–236. ACM Press, 1997.

26. N.D. Jones and S.S. Muchnick. Flow analysis and optimization of Lisp-like structures. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, Chapter 4, pp. 102-131, Prentice-Hall, 1981.
27. O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In: *POPL*, pp. 97–107. ACM Press, 1985.
28. G. Nelson. Verifying reachability invariants of linked structures. In: *POPL*, pp. 38–47. ACM Press, 1983.
29. R. Manevich, E. Yahav, G. Ramalingam, and M. Sagiv. Predicate abstraction and canonical abstraction for singly-linked lists. In: *VMCAI*, LNCS 3385, pp. 181–198, 2005.
30. R. Milner. *A Calculus of Communicating Systems*. LNCS 92, Springer, 1980.
31. U. Montanari and M. Pistore. An introduction to history-dependent automata. *ENTCS* **10**, 1998.
32. A. Møller and M. Schwartzbach. The pointer assertion logic engine. In: *PLDI*, pp. 221–213. ACM Press, 2001.
33. J. Morris. Assignment and linked data structures. In: *Th. Found. of Progr. Meth.*, pp. 25–34. Reidel, 1981.
34. P.W. O'Hearn, H. Yang, and J.C. Reynolds. Separation and information hiding. In: *POPL*, pp. 268–280. ACM Press, 2004.
35. A. Pnueli. The temporal logic of programs. In: *FOCS*, pp. 46–57. IEEE CS Press, 1977.
36. A. Rensink. Canonical graph shapes. In: *ESOP*, LNCS 2986, pp. 401–415. Springer, 2004.
37. J.C. Reynolds. Separation logic: A logic for shared mutable data structures. In: *LICS*, pp. 55–74. IEEE CS Press, 2002.
38. M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM TOPLAS*, **20**(1): 1–50, 1998.
39. L. Séméria, K. Sato and G. de Micheli. Resolution of dynamic memory allocation and pointers for the behavioural synthesis from `C`. In: *DATE*, pp. 312–319. ACM Press, 2000.
40. E. Yahav, T. Reps, M. Sagiv, and R. Wilhelm. Verifying temporal heap properties specified via evolution logic. In: *ESOP*, LNCS 2618, pp. 204–222. Springer, 2003.
41. T. Yavuz-Kahveci and T. Bultan. Automated verification of concurrent linked lists with counters. In: *SAS*, LNCS 2477, pp. 69–82, 2002.