# Heap Analysis and Verification

## HAV 2007

Informal Proceedings

March 25, 2007

Braga, Portugal

# Contents

James Brotherston
Proof Systems for Inductive Reasoning in the Logic of Bunched Implications ... 1

Thomas Wies, Viktor Kuncak, Karen Zee, Martin Rinard, Andreas Podelski
Verifying Complex Properties using Symbolic Shape Analysis ... 16

Yannick Moy, Claude Marché
Inferring Local (Non-)Aliasing and Strings for Memory Safety ... 35

Rémi Brochenin, Stéphane Demri, Etienne Lozes
Reasoning about sequences of memory states (preliminary version) ... 52

Amey Karkare, Uday Khedker, Amitabha Sanyal
Liveness of Heap Data for Functional Programs ... 64

Thierry Hubert, Claude Marché
Separation Analysis for Deductive Verification ... 81

Joost–Pieter Katoen, Thomas Noll, Stefan Rieger
Verifying Concurrent List-Manipulating Programs by LTL Model Checking ... 94

# Proof Systems for Inductive Reasoning in the Logic of Bunched Implications

James Brotherston[1],[2]

*Dept of Computing*
*Imperial College*
*London, UK*

**Abstract**

We report on our early progress in developing suitable frameworks for inductive reasoning in separation logic and related logics for low-level program verification, following the approach of our previous work on sequent proof systems for first-order logic with inductive definitions. We extend a first-order predicate version of the logic of bunched implications, BI — of which separation logic is a special instance — with a framework for inductively defined predicates in which both the additive and multiplicative conjunctions of BI may occur in the premises of definitions. We then formulate two Gentzen-type proof systems for this logic: a traditional finitary system incorporating explicit induction rules; and a system for *cyclic proof* in which the induction rules are replaced by weaker case-split rules, and proofs are derivation trees with "back edges", satisfying a global syntactic condition ensuring soundness. We illustrate our framework and proof systems with simple examples that indicate that, in our setting, cyclic proof may enjoy certain advantages over the traditional induction approach.

*Keywords:* inductive definitions, cyclic proof, logic of bunched implications, separation logic, substructural logics

## 1 Introduction

The mechanised verification of properties of computer programs — for example, properties expressing safety, liveness, or correctness — is an important and very challenging problem currently attracting considerable interest amongst researchers in computer science. A major source of inconvenience, though, is the fact that real-life computer programs tend to be written in low-level languages employing pointer arithmetic and similar operations that directly alter data stored in shared mutable structures, such as the heap. Because the (potentially dangerous) effects of these operations are hard to analyse, programs written using such languages have so far proven very much less amenable to formal reasoning than those written in, e.g., high-level functional programming languages, which are typically more well-behaved from a mathematical standpoint.

[1] This research is supported by EPSRC grant EP/E002536/1. My thanks also to Richard Bornat, Cristiano Calcagno, Peter O'Hearn, David Pym, and Hongseok Yang for helpful discussions, references and advice.

[2] Email: J.Brotherston@imperial.ac.uk

Inductive definitions are a well-established tool for representing many structures commonly used in the specification of computer programs, such as linked lists and binary trees. For any inductively defined structure, there are naturally associated inductive proof principles allowing us to reason about the structure by exploiting the recursion in its definition. Most often, these principles are encoded as inference rules or axioms in the native reasoning framework; but one can also reason with inductive definitions via a natural mode of *cyclic proof* [8,9]. In contrast to the usual finite tree proofs, cyclic proofs are regular infinite trees — represented as a finite (cyclic) graph — satisfying a global condition ensuring soundness. For inductively defined relations, this soundness condition is manifested as a generalisation of the principle of infinite descent *à la* Fermat. This paper reports on work in progress to develop suitable frameworks for reasoning about inductive definitions in low-level program specifications, including appropriate notions of cyclic proof.

The logic of *bunched implications* (BI), formulated by O'Hearn and Pym [18], offers a convenient formalism for expressing properties of programs that access and modify some shared, mutable resource or data structure [14]. The main feature of BI is that, as well as the usual additive conjunction ($\wedge$), it includes a multiplicative (or linear) conjunction, $*$, which is used to state that the current resource state can be decomposed into two disjoint parts in which its conjuncts respectively hold. This conjunction is accompanied by a suitably adjunctive implication $-\!\!*$, which expresses properties involving the addition of fresh resource to the current state. Because $*$ admits neither weakening ($P * Q \not\Rightarrow Q$) nor contraction ($P * P \not\Rightarrow P$), the logic BI belongs to the general class of *substructural logics* (see e.g. [20]).

By considering particular models of BI, one can obtain logics suitable for carrying out verification in specific programming languages. One useful such logic is Reynolds' *separation logic* — essentially obtained by taking a model of BI in which the domain of interpretation is given by a model of stacks and the resource states are given by a model of heaps — which is suitable for reasoning about C-like languages [21]. Separation logic has to date been used in the verification of several non-trivial programs involving pointer arithmetic, including (but not limited to) a copying garbage collector [6], a DAG duplication program [7] and the Schorr-Waite graph marking algorithm [23]. It has also fruitfully been employed in local shape analysis [12,13], program termination analysis [4], and automated program verification (see e.g. [3,2]).

As has already been noted by Biering *et al* [5], existing formal developments in separation logic have typically relied upon *ad hoc* extensions of the core logic by the inductive definitions needed for the development. It is thus of clear interest to develop a formal framework for separation logic in which one can define and reason about inductive structures. However, it is not desirable merely to add some sufficiently powerful principle (e.g. full second-order quantification) to the logic; our eventual goal is automated proof search, which is known to be already very difficult in the presence of induction even in elementary settings such as first-order arithmetic (see e.g. [11]), and so our aim will be to produce proof calculi that are amenable to proof search.

In Section 2, we extend a version of first-order predicate BI with a framework for (possibly mutual) inductively defined relations, based on simple "productions"

in the style of Martin-Löf [15], in which the multiplicative conjunction and identity may occur in the premises of definitions. This framework, though relatively simple, appears nonetheless powerful enough to express the inductive definitions that have arisen in practice in existing applications of separation logic to program verification. In Section 3 we extend the usual Gentzen-style proof system for BI to obtain a proof system supporting induction in the extended logic $\mathrm{BI}_{\mathrm{ID}}$ by adding left- and right-introduction rules for atomic formulas involving the inductive predicates of the theory. Following the approach taken in [8,9,15], the right introduction rules for an inductive predicate $P$ are merely sequent versions of the productions defining $P$, while the left-introduction rule for $P$ embodies the natural principle of rule induction over the definition of $P$. However, there is also a natural notion of cyclic proof for the logic, for which we introduce a second proof system in Section 4. In this system, the induction rules of the first system are replaced by simple *case-split* rules. Proofs in the system are "unfinished" derivation trees together with a function identifying every *bud* — nodes to which no proof rule has been applied — with an identical interior node called its *companion*; such proofs can straightforwardly be understood as cyclic graphs by identifying buds with their companions. In general, such proofs are not sound, so to ensure correctness we impose a condition stipulating, essentially, that for each infinite path in the proof, some inductive definition is unfolded infinitely often along the path. By a generalisation of Fermat's infinite descent principle to inductively defined relations, all such paths can be disregarded, whereby the remaining portion of proof is well-founded and hence sound. Finally, in Section 5, we identify the main directions for future work.

## 2 First-order predicate BI with inductive definitions

In this section we give the syntax and semantics of our logic, $\mathrm{BI}_{\mathrm{ID}}$, obtained by extending first-order predicate BI à la Biering *et al* [5] with a framework for (possibly mutual) inductive definitions. The latter is based on the framework from [8,9], which in turn is based on Martin-Löf's "ordinary productions" [15], except that we allow the multiplicative conjunction and identity of BI to occur in the premises of definitions.

A brief comment on some of our mathematical and notational conventions is in order. We often use vector notation to abbreviate sequences, e.g. $\mathbf{x}$ for $(x_1, \ldots, x_n)$; for any $n \in \mathbb{N}$ and $i \leq n$ we define the $i$th projection function $\pi_i^n$ on $n$-tuples of sets by $p_i^n(X_1, \ldots, X_n) = X_i$; for any $n \in \mathbb{N}$ we extend set union, intersection and inclusion to $n$-tuples of sets by their corresponding pointwise definitions.

Our languages are the standard (countable) first-order languages — containing arbitrarily many constant, function, and predicate symbols — except that we designate finitely many of the predicate symbols as *inductive*. A predicate symbol that is not inductive is called *ordinary*. For the rest of this paper, we shall consider a fixed language $\Sigma$ containing exactly $n$ inductive predicates $P_1, \ldots, P_n$. We also assume the existence of a denumerably infinite set $\mathcal{V}$ of variables, each of which is distinct from any symbol in $\Sigma$.

The elements of $\Sigma$ are as usual interpreted by a structure, with the difference here that our structures include a notion of a set of possible resource states or

"worlds", given by a partial commutative monoid. The interpretation of predicates is parameterised by the elements of this monoid: in other words, the set of (tuples of) objects in the domain of which a given predicate is true depends on the current resource state. (However, the interpretations of the constant and function symbols are independent of the resource state.)

**Definition 2.1** [BI-structure] A BI-*structure* for $\Sigma$ is a tuple:

$$M = (D, \langle R, \circ, e\rangle, c_1^M, c_2^M, \ldots, f_1^M, f_2^M, \ldots, Q_1^M, Q_2^M, \ldots, P_1^M, \ldots, P_n^M)$$

where $D$ is a set (called the *domain* of $M$), $\langle R, \circ, e\rangle$ is a partial commutative monoid and:

- each $c_i^M \in D$;
- each $f_i^M : D^k \to D$, where $k$ is the arity of the function symbol $f_i$;
- each $Q_i^M \subseteq R \times D^k$, where $k$ is the arity of the ordinary predicate symbol $Q_i$;
- each $P_i^M \subseteq R \times D^k$, where $k$ is the arity of the inductive predicate symbol $P_i$;

Although our structures interpret the inductive predicate symbols of $\Sigma$, this is purely for technical convenience: we shall only be interested later in those structures in which the interpretation of the inductive predicates coincides with our intended interpretation, given by a fixed set of inductive definitions.

The *terms* of $\Sigma$ are defined as usual;we write $t[u/x]$ to denote the term obtained by substituting the term $u$ for all occurrences of the variable $x$ in the term $t$. We write $Var(t)$ for the set of variables appearing in the term $t$, and write $t(x_1, \ldots, x_n)$ for a term $t$ such that $Var(t) \subseteq \{x_1, \ldots, x_n\}$, where $x_1, \ldots, x_n$ are distinct. In this case we may write $t(t_1, \ldots, t_n)$ to denote the term obtained by substituting $t_1, \ldots, t_n$ for $x_1, \ldots, x_n$ respectively in $t$. Also, $t^M(x_1, \ldots, x_k) : D^k \to D$ is obtained by replacing every constant symbol $c$ by $c^M$ and every function symbol $f$ by $f^M$ in $t(x_1, \ldots, x_n)$.

The formulas of $\mathrm{BI_{ID}}$ are just the standard formulas of predicate BI [3], given by the following definition:

**Definition 2.2** [BI-formulas] The set of $\Sigma$-formulas of BI is the smallest set of expressions closed under the following rules:

- $\top$, $\bot$ and $I$ are atomic formulas;
- if $t_1, \ldots, t_k$ are terms of $\Sigma$, and $Q$ is a predicate symbol in $\Sigma$ of arity $k$, then $Q(t_1, \ldots, t_k)$ is an atomic formula;
- if $t$ and $u$ are terms of $\Sigma$ then $t = u$ is an atomic formula;
- if $F_1$ and $F_2$ are formulas then $F_1 \wedge F_2$, $F_1 \vee F_2$, $F_1 \to F_2$, $F_1 * F_2$ and $F_1 \mathrel{-\!\!*} F_2$ are (non-atomic) formulas;
- if $F$ is a formula and $x \in \mathcal{V}$ is a variable, then $\exists x F$ and $\forall x F$ are (non-atomic) formulas.

We use the standard precedences on the logical connectives — with $*$ and $\mathrel{-\!\!*}$ having the same logical precedence as $\wedge$ and $\to$ respectively — and use parentheses to

[3] As in [5], our "predicate BI" is propositional BI extended with the usual additive quantifiers $\forall$ and $\exists$, as opposed to propositional BI extended with both additive and multiplicative versions of the quantifiers, as in e.g. [18].

disambiguate where necessary. Also, we write $F \leftrightarrow G$ to abbreviate $(F \rightarrow G) \wedge (G \rightarrow F)$, and $\neg F$ to abbreviate $F \rightarrow \bot$.

As in first-order logic, we interpret variables as elements of the domain $D$ of a BI-structure using environments $\rho : \mathcal{V} \rightarrow D$; we extend environments to all terms of $\Sigma$ in the usual way and write $\rho[x \mapsto d]$ for the environment defined exactly as $\rho$ except that $\rho(x) = d$. The formulas of $\mathrm{BI}_{\mathrm{ID}}$ are then interpreted by the following satisfaction (a.k.a. "forcing") relation:

**Definition 2.3** [Satisfaction relation for BI] Let $M = (D, \langle R, \circ, e \rangle, \ldots)$ be a BI-structure for the language $\Sigma$, let $r \in R$ and let $\rho$ be an environment for $M$. We define the satisfaction relation $M, r \models_\rho F$ on formulas by:

$$\begin{array}{rcl}
M, r \models_\rho \top & \Leftrightarrow & \text{true} \\
M, r \models_\rho \bot & \Leftrightarrow & \text{false} \\
M, r \models_\rho I & \Leftrightarrow & r = e \\
M, r \models_\rho Q\mathbf{t} & \Leftrightarrow & Q^M(r, \rho(\mathbf{t})) \quad (Q \text{ ordinary or inductive}) \\
M, r \models_\rho t = u & \Leftrightarrow & \rho(t) = \rho(u) \\
M, r \models_\rho F_1 \wedge F_2 & \Leftrightarrow & M, r \models_\rho F_1 \text{ and } M, r \models_\rho F_2 \\
M, r \models_\rho F_1 \vee F_2 & \Leftrightarrow & M, r \models_\rho F_1 \text{ or } M, r \models_\rho F_2 \\
M, r \models_\rho F_1 \rightarrow F_2 & \Leftrightarrow & M, r \models_\rho F_1 \text{ implies } M, r \models_\rho F_2 \\
M, r \models_\rho F_1 * F_2 & \Leftrightarrow & r = r_1 \circ r_2 \text{ and } M, r_1 \models_\rho F_1 \text{ and } M, r_2 \models_\rho F_2 \\
 & & \text{for some } r_1, r_2 \in R \\
M, r \models_\rho F_1 \mathrel{-\!\!*} F_2 & \Leftrightarrow & M, r' \models_\rho F_1 \text{ implies } M, r' \circ r \models_\rho F_2 \text{ for all } r' \in R \\
M, r \models_\rho \forall x F & \Leftrightarrow & M, r \models_{\rho[x \mapsto d]} F \text{ for all } d \in D \\
M, r \models_\rho \exists x F & \Leftrightarrow & M, r \models_{\rho[x \mapsto d]} F \text{ for some } d \in D
\end{array}$$

(Informally, $M, r \models_\rho F$ means: "the formula $F$ is true in $M$ in the resource state $r$ and under the environment $\rho$".)

We now give our schema for (possibly mutual) inductive definitions, which is based on Martin-Löf's schema for "ordinary productions" [15]. However, here our premises are not merely lists of atomic formulas, but clauses in which atomic formulas and the multiplicative identity formula $I$ may freely be combined using the additive and multiplicative conjunctions of BI:

**Definition 2.4** [Inductive definition set] An *inductive definition set* for $\Sigma$ is a set of *productions*, which are rules of the form:

$$\frac{C(\mathbf{x})}{P_i \mathbf{t}(\mathbf{x})} \quad i \in \{1, \ldots, n\}$$

where $C(\mathbf{x})$ is a *combinative inductive clause* given by the following grammar:

$$C(\mathbf{x}) ::= I \mid Q\mathbf{t}(\mathbf{x}) \mid P_i\mathbf{t}(\mathbf{x}) \ (i \in \{1, \ldots, n\}) \mid C(\mathbf{x}) \wedge C(\mathbf{x}) \mid C(\mathbf{x}) * C(\mathbf{x})$$

where $Q$ ranges over the ordinary predicate symbols of $\Sigma$.

The productions whose conclusions feature an inductive predicate $P$ should be considered as disjunctive clauses of the definition of $P$. For some readers the following, equivalent notation for definitions may be more familiar:

$$P\mathbf{x} =_{def} (\mathbf{x} = \mathbf{t_1}(\mathbf{y}) \wedge C_1(\mathbf{y})) \vee \ldots \vee (\mathbf{x} = \mathbf{t_k}(\mathbf{y})) \wedge C_k(\mathbf{y}))$$

where $C_1(\mathbf{y}), \ldots, C_k(\mathbf{y})$ are combinative inductive clauses. It is trivial to convert from either form to the other.

As usual, the intended interpretation of the inductive predicate symbols of $\Sigma$ is obtained by taking the least fixed point of a monotone operator constructed from the definition set $\Phi$. Our parameterisation of predicate interpretations by resource states, and the corresponding use of $*$ and $I$ in our inductive definitions, entails some extra complication in the construction of this operator, so we spell out the details:

**Definition 2.5** [Definition set operator] Let $\Sigma$ be a language with exactly $n$ inductive predicates $P_1, \ldots, P_n$, let $M = (D, \langle R, \circ, e\rangle, \ldots)$ be a BI-structure for $\Sigma$, let $\Phi$ be an inductive definition set for $\Sigma$ and, for each $i \in \{1, \ldots, n\}$, let $k_i$ be the arity of the inductive predicate symbol $P_i$. We partition $\Phi$ into disjoint subsets $\Phi_1, \ldots, \Phi_n \subseteq \Phi$ by: $\Phi_i = \{\mathit{Prod} \in \Phi \mid P_i \text{ occurs in the conclusion of } \mathit{Prod}\}$. We let each definition set $\Phi_i$ be indexed by $j$ with $j \in \{1, \ldots, |\Phi_i|\}$, and from each production $\Phi_{i,j} \in \Phi_i$, say:

$$\frac{C(\mathbf{x})}{P_i\mathbf{t}(\mathbf{x})}$$

we obtain a corresponding function $\varphi_{i,j}$ as follows:

$$\varphi_{i,j}(X_1, \ldots, X_n) = \{(r, \mathbf{t}^M(\mathbf{d})) \mid T^r(C(\mathbf{d}))(X_1, \ldots, X_n)\}$$

where the relation $T^r$ is defined inductively on the structure of combinative inductive clauses as follows:

$$\begin{aligned}
T^r(I)(X_1, \ldots, X_n) &\Leftrightarrow r = e \\
T^r(Q\mathbf{t}(\mathbf{d}))(X_1, \ldots, X_n) &\Leftrightarrow Q^M\mathbf{t}^M(\mathbf{d}) \\
T^r(P_i\mathbf{t}(\mathbf{d}))(X_1, \ldots, X_n) &\Leftrightarrow \mathbf{t}^M(\mathbf{d}) \in X_i \\
T^r(C_1(\mathbf{d}) \wedge C_2(\mathbf{d}))(X_1, \ldots, X_n) &\Leftrightarrow T^r(C_1(\mathbf{d}))(X_1, \ldots, X_n) \\
&\quad\;\; \text{and } T^r(C_2(\mathbf{d}))(X_1, \ldots, X_n) \\
T^r(C_1(\mathbf{d}) * C_2(\mathbf{d}))(X_1, \ldots, X_n) &\Leftrightarrow \exists r_1, r_2.\ (r = r_1 \circ r_2 \text{ and} \\
&\quad\;\; T^{r_1}(C_1(\mathbf{d}))(X_1, \ldots, X_n) \text{ and} \\
&\quad\;\; T^{r_2}(C_2(\mathbf{d}))(X_1, \ldots, X_n))
\end{aligned}$$

(Note that any variables occurring in the right hand side but not the left hand side of the set expression in the definition of $\varphi_{i,j}$ above are, implicitly, existentially quantified over the entire right hand side of the expression.) Then the *definition set operator* for $\Phi$ is the operator $\varphi_\Phi$, with domain and codomain $\mathrm{Pow}(R \times D^{k_1}) \times$

$\ldots \times \text{Pow}(R \times D^{k_n})$, defined by:

$$\varphi_\Phi(X_1,\ldots,X_n) = (\bigcup_j \varphi_{1,j}(X_1,\ldots,X_n),\ldots,\bigcup_j \varphi_{n,j}(X_1,\ldots,X_n))$$

**Proposition 2.6** *The operator $\varphi_\Phi$ is monotone (with respect to $\subseteq$).*

**Example 2.7** Let $\Phi_N$ be the inductive definition set consisting of the following productions for a unary inductive predicate $N$:

$$\frac{}{N0} \qquad \frac{Nx}{Nsx}$$

Then the definition set operator for $\Phi_N$ is defined by:

$$\varphi_{\Phi_N}(X) = \{(r, 0^M) \mid r \in R\} \cup \{(r, s^M d) \mid (r,d) \in X\}$$

In structures $M$ in which all "numerals" $(s^M)^k 0^M$ for $k \geq 0$ are distinct, the predicate $N$ corresponds to the property of being a natural number.

**Example 2.8** Let $\mapsto$ be a ordinary, binary predicate symbol (written infix), and let $\Phi_{ls}$ be the inductive definition set consisting of the following productions for a binary inductive predicate $\texttt{ls}$:

$$\frac{I}{\texttt{ls}\,x\,x} \qquad \frac{x \mapsto x' * \texttt{ls}\,x'\,y}{\texttt{ls}\,x\,y}$$

Then the definition set operator for $\Phi_{ls}$ is defined by:

$$\begin{aligned}\varphi_{\Phi_{ls}}(X) = \{&(e,(d,d)) \mid d \in D\} \\ &\cup\{(r_1 \circ r_2,(d,d')) \mid (r_1,(d,d'')) \in \mapsto^M \text{ and } (r_2,(d'',d')) \in X\}\end{aligned}$$

where $d''$ in the second set comprehension is, implicitly, existentially quantified. In separation logic, where the resource states are heaps and $x \mapsto y$ means "$x$ is a pointer to $y$", the predicate $\texttt{ls}$ is used to represent list segments, so that $\texttt{ls}\,x\,y$ is true of a particular heap $h$ if $h$ represents a linked list whose first element is pointed to by $x$ and whose last element contains a dangling pointer $y$.

It is a standard result for inductive definitions that the least $n$-tuple of sets closed under the productions in $\Phi$ is the least prefixed point of the operator $\varphi_\Phi$ (see e.g. [1,16]), and that this least prefixed point can be approached in iterative *approximant* stages, as follows:

**Definition 2.9** [Approximants] Let $\Phi$ be an inductive definition set for $\Sigma$, .and define a chain of ordinal-indexed sets $(\varphi^\alpha_\Phi)_{\alpha\geq 0}$ by transfinite induction: $\varphi^\alpha_\Phi = \bigcup_{\beta<\alpha} \varphi_\Phi(\varphi^\beta_\Phi)$ (note that this implies $\varphi^0_\Phi = (\emptyset,\ldots,\emptyset)$). Then for each $i \in \{1,\ldots,n\}$, the set $\pi^n_i(\varphi^\alpha_\Phi)$ is called the $\alpha^{th}$ *approximant of* $P_i$, written as $P^\alpha_i$.

**Definition 2.10** [Standard model] Let $\Phi$ be an inductive definition set for $\Sigma$. Then a BI-structure $M$ for $\Sigma$ is said to be a *standard model* for $(\Sigma,\Phi)$ if $P^M_i = \bigcup_\alpha P^\alpha_i$ for all $i \in \{1,\ldots,n\}$.

Definition 2.10 thus fixes within a BI-structure a standard interpretation of the inductive predicate symbols of $\Sigma$ that is uniquely determined by the other components of the structure.

## 3 A proof system for induction in $\mathrm{BI}_{\mathrm{ID}}$

In this section we give a Gentzen-style proof system suitable for formalising traditional proof by induction in our logic $\mathrm{BI}_{\mathrm{ID}}$. Our starting point will be the standard sequent calculus for BI (cf. [19]).

We write *sequents* of the form $\Gamma \vdash F$, where $F$ is a formula and $\Gamma$ is a *bunch*, given by the following definition:

**Definition 3.1** [Bunch] A *bunch* is a tree whose leaves are labelled by formulas of $\mathrm{BI}_{\mathrm{ID}}$ and whose internal nodes are labelled by ';' or ',' (denoting respectively additive and multiplicative combination[4])

As our sequents have at most one formula occurring on the right hand side, our proof system is intuitionistic. This is not for ideological reasons but for technical convenience; the formulation of a classical (multiple-conclusion) sequent calculus for BI would necessitate the use of a multiplicative disjunction (for details see [19]).

We write $\Gamma(\Delta)$ to mean that $\Gamma$ is a bunch of which $\Delta$ is a subtree (called a "sub-bunch"), and write $\Gamma(\Delta')$ for the bunch obtained by replacing $\Delta$ by $\Delta'$ in $\Gamma(\Delta)$.

**Definition 3.2** [Coherent equivalence for bunches] Define $\equiv$ to be the least relation on bunches satisfying:

(i) commutative monoid equations for ';' and $\top$ (i.e., equations expressing that ';' is associative and commutative with respect to $\equiv$, and that $\top$ is the unit of ';' with respect to $\equiv$);

(ii) commutative monoid equations for ',' and $I$;

(iii) congruence: if $\Delta \equiv \Delta'$ then $\Gamma(\Delta) \equiv \Gamma(\Delta')$.

The usual sequent calculus rules for our version of predicate BI, plus rules for equality and an explicit substitution rule, are given in Figure 1. Our proof system, called $\mathrm{LBI}_{\mathrm{ID}}$, is obtained from this system by adding rules for introducing atomic formulas of the form $P_i\mathbf{t}$, where $P_i$ is an inductive predicate symbol, on the left and right of sequents.

First, for each production $\Phi_{i,j} \in \Phi$, we obtain a right-introduction rule $(P_iR_j)$ for the predicate $P_i$ as follows:

$$\frac{C(\mathbf{x})}{P_i\mathbf{t}(\mathbf{x})} \quad \Longrightarrow \quad \frac{\Gamma \vdash C(\mathbf{u})}{\Gamma \vdash P_i\mathbf{t}(\mathbf{u})}\,(P_iR_j)$$

Before giving the rules for introducing inductive predicates on the left of sequents, we first give a formal definition of what it means for two inductive predicates to have a mutual definition in $\Phi$ (repeated from [8]):

**Definition 3.3** [Mutual dependency] Define the binary relation *Prem* on the inductive predicate symbols $\{P_1, \ldots, P_n\}$ of $\Sigma$ as the least relation satisfying: whenever $P_i$ occurs in the conclusion of some production $\Phi_{i,r} \in \Phi$, and $P_j$ occurs in the

[4] Confusingly, although ',' is standardly used for additive combination in traditional sequent calculus, it is used to mean multiplicative combination in substructural logic, and we retain the latter usage for consistency with other formulations of BI. The original definition of a bunch also allowed an additive unit $\{\}_a$, equivalent to the formula $\top$, and a multiplicative unit $\{\}_m$, equivalent to the formula $I$, to occur in leaf positions; for our present purposes, this is an obvious redundancy.

**Structural rules:**

$$\frac{}{\Gamma; F \vdash F}\,(\text{Id}) \qquad \frac{\Gamma(\Delta) \vdash F}{\Gamma(\Delta; \Delta') \vdash F}\,(\text{Weak}) \qquad \frac{\Gamma(\Delta; \Delta) \vdash F}{\Gamma(\Delta) \vdash F}\,(\text{Contr})$$

$$\frac{\Gamma' \vdash F}{\Gamma \vdash F}\ \Gamma \equiv \Gamma'\ (\text{Equiv}) \qquad \frac{\Delta \vdash G \quad \Gamma(G) \vdash F}{\Gamma(\Delta) \vdash F}\,(\text{Cut}) \qquad \frac{\Gamma \vdash F}{\Gamma[\theta] \vdash F[\theta]}\,(\text{Subst})$$

**Propositional rules:**

$$\frac{}{\bot \vdash F}\,(\bot\text{L}) \qquad \frac{\Gamma(F_1; F_2) \vdash F}{\Gamma(F_1 \wedge F_2) \vdash F}\,(\wedge\text{L}) \qquad \frac{\Gamma(F_1) \vdash F \quad \Gamma(F_2) \vdash F}{\Gamma(F_1 \vee F_2) \vdash F}\,(\vee\text{L})$$

$$\frac{}{\Gamma \vdash \top}\,(\top\text{R}) \qquad \frac{\Gamma \vdash F_1 \quad \Gamma \vdash F_2}{\Gamma \vdash F_1 \wedge F_2}\,(\wedge\text{R}) \qquad \frac{\Gamma \vdash F_i}{\Gamma \vdash F_1 \vee F_2}\ i \in \{1, 2\}\ (\vee\text{R})$$

$$\frac{\Delta \vdash F_1 \quad \Gamma(\Delta', F_2) \vdash F}{\Gamma(\Delta, \Delta', F_1 -\!\!* F_2) \vdash F}\,(-\!\!*\text{L}) \qquad \frac{\Delta \vdash F_1 \quad \Gamma(\Delta; F_2) \vdash F}{\Gamma(\Delta; F_1 \to F_2) \vdash F}\,(\to\text{L}) \qquad \frac{\Gamma(F_1, F_2) \vdash F}{\Gamma(F_1 * F_2) \vdash F}\,(*\text{L})$$

$$\frac{\Gamma, F_1 \vdash F_2}{\Gamma \vdash F_1 -\!\!* F_2}\,(-\!\!*\text{R}) \qquad \frac{\Gamma; F_1 \vdash F_2}{\Gamma \vdash F_1 \to F_2}\,(\to\text{R}) \qquad \frac{\Gamma \vdash F_1 \quad \Delta \vdash F_2}{\Gamma, \Delta \vdash F_1 * F_2}\,(*\text{R})$$

**Quantifier rules:**

$$\frac{\Gamma(G[t/x]) \vdash F}{\Gamma(\forall x G) \vdash F}\,(\forall\text{L}) \qquad \frac{\Gamma \vdash F}{\Gamma \vdash \forall x F}\ x \notin FV(\Gamma \cup \{F\})\ (\forall\text{R})$$

$$\frac{\Gamma(G) \vdash F}{\Gamma(\exists x G) \vdash F}\ x \notin FV(\Gamma \cup \{F\})\ (\exists\text{L}) \qquad \frac{\Gamma \vdash F[t/x]}{\Gamma \vdash \exists x F}\,(\exists\text{R})$$

**Equality rules:**

$$\frac{}{\Gamma \vdash t = t}\,(=\text{R}) \qquad \frac{\Gamma[u/x, t/y] \vdash F[u/x, t/y]}{\Gamma(t = u)[t/x, u/y] \vdash F[t/x, u/y]}\,(=\text{L})$$

Fig. 1. Sequent calculus proof rules for predicate BI with equality.

premise of that production, then $Prem(P_i, P_j)$ holds. Also define $Prem^*$ to be the reflexive-transitive closure of $Prem$. Then we say two predicate symbols $P$ and $Q$ are *mutually dependent* if both $Prem^*(P, Q)$ and $Prem^*(Q, P)$ hold.

Now to obtain an instance of the induction rule for any inductive predicate $P_j$, we first associate with every inductive predicate $P_i$ a tuple $\mathbf{z_i}$ of $k_i$ distinct variables (called *induction variables*), where $k_i$ is the arity of $P_i$. Furthermore, we associate to every predicate $P_i$ that is mutually dependent with $P_j$ a formula (called an *induction hypothesis*) $H_i$, possibly containing some of the induction variables. Next, define the formula $G_i$ for each $i \in \{1, \ldots, n\}$ by: $G_i = H_i$ if $P_i$ and $P_j$ are mutually dependent, and $G_i = P_i\mathbf{z_i}$ otherwise. For convenience, we shall write $G_i\mathbf{t}$ for $G_i[\mathbf{t}/\mathbf{z_i}]$, where $\mathbf{t}$ is a tuple of $k_i$ terms. Then an instance of the induction rule (Ind $P_j$) for $P_j$ has the following schema:

$$\frac{\text{minor premises} \qquad \Gamma(\Delta; H_j\mathbf{t}) \vdash F}{\Gamma(\Delta; P_j\mathbf{t}) \vdash F}\,(\text{Ind } P_j)$$

where the premise $\Gamma, H_j\mathbf{t} \vdash \Delta$ is called the *major premise* of the rule, and for each production of $\Phi$ having in its conclusion a predicate $P_i$ that is mutually dependent

with $P_j$, we obtain a *minor premise* as follows:

$$\frac{C(\mathbf{x})}{P_i\mathbf{t}(\mathbf{x})} \quad \Longrightarrow \quad \Delta; C_H(\mathbf{x}) \vdash H_i\mathbf{t}(\mathbf{x}) \quad (\forall x \in \mathbf{x}.\, x \notin FV(\Delta))$$

where $C_H(\mathbf{x})$ is the formula obtained by replacing every formula of the form $P_k\mathbf{t}(\mathbf{x})$ (for $P_k$ an inductive predicate) by $G_k\mathbf{t}(\mathbf{x})$ in the combinative inductive clause $C(\mathbf{x})$.

**Example 3.4** The induction rule for the predicate $N$ from example 2.7 is:

$$\frac{\Delta \vdash H0 \quad \Delta; Hx \vdash Hsx \quad \Gamma(\Delta; Ht) \vdash F}{\Gamma(\Delta; Nt) \vdash F}\ (\text{Ind } N)$$

where $H$ is the induction hypothesis associated with $N$ and $x$ is suitably fresh.

**Example 3.5** The induction rule for the predicate `ls` from example 2.8 is:

$$\frac{\Delta; I \vdash Hxx \quad \Delta; x \mapsto x' * Hx'y \vdash Hxy \quad \Gamma(\Delta; Htu) \vdash F}{\Gamma(\Delta; \mathtt{ls}\, t\, u) \vdash F}\ (\text{Ind } \mathtt{ls})$$

where $H$ is the induction hypothesis associated with `ls` and $x, x', y$ are fresh.

**Definition 3.6 (Validity)** *Let $M$ be a standard model for $(\Sigma, \Phi)$. Then a sequent $\Gamma \vdash F$ is said to be* true *in $M$ if for all environments $\rho$ and resource states $r$, $M, r \models_\rho \phi_\Gamma$ implies $M, r \models_\rho F$, where $\phi_\Gamma$ is the formula obtained by replacing every occurrence of ';' by $\wedge$ and every occurrence of ',' by $*$ in the bunch $\Gamma$. $\Gamma \vdash F$ is said to be* valid *if it is true in all standard models of $(\Sigma, \Phi)$.*

By a *derivation tree*, we mean a finite tree of sequents in which each parent sequent is obtained as the conclusion of an inference rule with its children as premises. We distinguish between "leaves" and "buds" in the tree. By a *leaf* we mean an axiom, i.e. the conclusion of a 0-premise inference rule. By a *bud* we mean any sequent occurrence in the tree that is not the conclusion of a proof rule. An $\mathrm{LBI_{ID}}$ *proof* is then, as usual, a finite derivation tree constructed according to the proof rules that contains no buds. The following proposition is a straightforward consequence of the local soundness of our proof rules.

**Proposition 3.7 (Soundness of $\mathbf{LBI_{ID}}$)** *If there is an $LBI_{ID}$ proof of $\Gamma \vdash \Delta$ then $\Gamma \vdash \Delta$ is valid.*

**Example 3.8** We give an $\mathrm{LBI_{ID}}$ proof that the predicate $N$ from Example 2.7 admits multiplicative weakening, i.e. that $F, Nx \vdash Nx$:

$$\dfrac{\dfrac{\dfrac{}{F \vdash N0}\,(NR_1)}{\vdash F \mathrel{-\!\!*} N0}\,(\mathrel{-\!\!*}\mathrm{R}) \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{F \vdash F}\,(\mathrm{Id}) \quad \dfrac{}{Ny \vdash Ny}\,(\mathrm{Id})}{F, F \mathrel{-\!\!*} Ny \vdash Ny}\,(\mathrel{-\!\!*}\mathrm{L})}{F, F \mathrel{-\!\!*} Ny \vdash Nsy}\,(NR_2)}{F \mathrel{-\!\!*} Ny \vdash F \mathrel{-\!\!*} Nsy}\,(\mathrel{-\!\!*}\mathrm{R}) \qquad \dfrac{\dfrac{}{F \vdash F}\,(\mathrm{Id}) \quad \dfrac{}{Nx \vdash Nx}\,(\mathrm{Id})}{F, F \mathrel{-\!\!*} Nx \vdash Nx}\,(\mathrel{-\!\!*}\mathrm{L})}{F, Nx \vdash Nx}\,(\text{Ind } N)$$

Note that in the application of (Ind $N$) in this proof we associate the induction variable $z$ and the induction hypothesis $F \mathrel{-\!\!*} Nz$ with the inductive predicate $N$. We remark that one can easily see that this example demonstrates the need for generalisation of induction hypotheses in this setting (at least for cut-free proofs):

it is clear that using either $F$ or $Nz$ as the induction hypothesis will not enable us to prove the major premise of the induction application.

## 4 A cyclic proof system for $\mathrm{BI}_{\mathrm{ID}}$

We now define a second proof system $\mathrm{CLBI}^{\omega}_{\mathrm{ID}}$ for $\mathrm{BI}_{\mathrm{ID}}$ which admits a notion of cyclic proof; our proof structures are finite derivation trees together with a function assigning to every unexpanded node in the proof tree (called a *bud*) an interior node with an identical sequent labelling (the *companion* of the bud). These structures (called *pre-proofs*) can then be viewed as cyclic graphs; we impose a global condition on pre-proofs to ensure soundness.

The proof rules of the system $\mathrm{CLBI}^{\omega}_{\mathrm{ID}}$ are the rules of $\mathrm{LBI}_{\mathrm{ID}}$ described in Section 3, except that for each inductive predicate $P_i$ of $\Sigma$, the induction rule (Ind $P_i$) of $\mathrm{LBI}_{\mathrm{ID}}$ is replaced by the *case-split rule*:

$$\frac{\text{case distinctions}}{\Gamma(P_i\mathbf{u}) \vdash F}\ (\text{Case } P_i)$$

where for each production having predicate $P_i$ in its conclusion, we obtain a corresponding case distinction as follows:

$$\frac{C(\mathbf{x})}{P_i\mathbf{t}(\mathbf{x})} \quad \Longrightarrow \quad \Gamma(\mathbf{u} = \mathbf{t}(\mathbf{x}); C_B(\mathbf{x})) \vdash F \quad (\forall x \in \mathbf{x}.\, x \notin FV(\Gamma \cup \{F\}))$$

where $C_B(\mathbf{x})$ is the bunch obtained from the formula $C(\mathbf{x})$ by replacing each occurrence of $\wedge$ by ';' and each occurrence of $*$ by ','.

Any atomic formula containing an inductive predicate symbol occurring in $C(\mathbf{x})$ in the case distinction above is said to be a *case-descendant* of the active formula $P_i\mathbf{u}$ of the rule instance.

**Example 4.1** The case-split rule for $N$ from Example 2.7 is:

$$\frac{\Gamma(t = 0) \vdash F \qquad \Gamma(t = sx; Nx) \vdash F}{\Gamma(Nt) \vdash F}\ (\text{Case } N)$$

**Example 4.2** The case-split rule for `ls` from Example 2.8 is:

$$\frac{\Gamma(t = u; I) \vdash F \qquad \Gamma(t \mapsto x, \mathtt{ls}\, x\, u) \vdash F}{\Gamma(\mathtt{ls}\, t\, u) \vdash F}\ (\text{Case } \mathtt{ls})$$

**Definition 4.3 (Companion)** *Let $B$ be a bud of a derivation tree $\mathcal{D}$. An internal node $C$ in $\mathcal{D}$ is said to be a* companion *for $B$ if they have the same sequent labelling.*

By assigning a companion to each bud node in a finite derivation tree, one obtains a finite representation of an associated (regular) infinite tree:

**Definition 4.4 ($\mathrm{CLBI}^{\omega}_{\mathrm{ID}}$ pre-proof)** *A* $\mathrm{CLBI}^{\omega}_{\mathrm{ID}}$ pre-proof *of a sequent $\Gamma \vdash \Delta$ is a pair $\mathcal{P} = (\mathcal{D}, \mathcal{R})$, where $\mathcal{D}$ is a derivation tree constructed according to the proof rules of $\mathrm{CLBI}^{\omega}_{\mathrm{ID}}$ given above such that $\Gamma \vdash \Delta$ appears at the root of $\mathcal{D}$, and $\mathcal{R}$ is a function assigning a companion to every bud of $\mathcal{D}$.*

*The* graph *of $\mathcal{P}$ is the graph $\mathcal{G}_{\mathcal{P}}$ obtained from $\mathcal{D}$ by identifying each bud node $B$ in $\mathcal{D}$ with its companion $\mathcal{R}(B)$.*

We observe that the local soundness of our proof rules is not sufficient to guarantee that pre-proofs are sound, due to the (possible) cyclicity evident in their graph representations. In order to give a criterion for soundness, we formulate the notion of a *trace* following a path in a pre-proof graph, similar to that used in [8,22] but more complex due to our use of bunches in sequents.

**Definition 4.5 (Trace)** *Let $\mathcal{P}$ be a $CLBI^{\omega}_{ID}$ pre-proof and let $(\Gamma_i \vdash F_i)$ be a path in $\mathcal{G}_\mathcal{P}$. A* trace following $(\Gamma_i \vdash F_i)$ *is a sequence $(\tau_i)$ such that, for all $i$:*

- *$\tau_i = P_i\mathbf{t_i}$ is a leaf of $\Gamma_i$, where $P_i$ is an inductive predicate;*
- *if $\Gamma_i \vdash F_i$ is the conclusion of (Subst) then $\tau_i = \tau_{i+1}[\theta]$, where $\theta$ is the substitution associated with the rule instance;*
- *if $\Gamma_i \vdash F_i$ is the conclusion of (=L) with active formula $t = u$ then there is a formula $G$ and variables $x, y$ such that $\tau_i = G[t/x, u/y]$ and $\tau_{i+1} = G[u/x, t/y]$;*
- *if $\Gamma_i \vdash \Delta_i$ is the conclusion of a case-split rule then either $\tau_{i+1} = \tau_i$, or $\tau_i$ is the active formula of the rule instance and $\tau_{i+1}$ is a case-descendant of $\tau_i$. In the latter case, $i$ is said to be a* progress point *of the trace;*
- *if $\Gamma_i \vdash \Delta_i$ is the conclusion of any other rule then $\tau_{i+1} = \tau_i$;*
- *the position of $\tau_{i+1}$ in $\Gamma_{i+1}$ is the same as the position of $\tau_i$ in $\Gamma_i$, modulo any splitting of $\Gamma_i$. E.g. if $\Gamma_i \vdash F_i$ is the conclusion of the inference:*

$$\frac{\Delta \vdash F_1 \quad \Gamma(\Delta', F_2) \vdash F}{\Gamma(\Delta, \Delta', F_1 \mathrel{-\!\!*} F_2) \vdash F}\ (\mathrel{-\!\!*}L)$$

  *then, if $\Gamma_{i+1} \vdash F_i$ is the left hand premise, we must have $\Delta = \Psi(\tau_i)$ and $\Delta = \Psi(\tau_{i+1})$ (for some $\Psi$). If $\Gamma_{i+1} \vdash F_i$ is the right hand premise, then we must have either $\Delta' = \Psi(\tau_i)$ and $\Delta' = \Psi(\tau_{i+1})$ or $\Gamma(-) = \Psi(\tau_i)$ and $\Gamma(-) = \Psi(\tau_{i+1})$.*

*An* infinitely progressing trace *is a trace having infinitely many progress points.*

Informally, a trace follows (a part of) the construction of an inductively defined predicate occurring in the left hand side of the sequents occurring on a path in a pre-proof. These predicate constructions never become larger as we follow the trace along the path, and at progress points, they actually decrease. This property is encapsulated in the following lemma and motivates the subsequent definition of a *cyclic proof*:

**Lemma 4.6** *Let $\mathcal{P}$ be a $CLBI^{\omega}_{ID}$ pre-proof of $\Gamma_0 \vdash F_0$, and let $M$ be a standard model such that $\Gamma_0 \vdash F_0$ is false in $M$ in the resource state $r_0$ and environment $\rho_0$ (say). Then there is an infinite path $(\Gamma_i \vdash F_i)_{i \geq 0}$ in $\mathcal{G}_\mathcal{P}$ and infinite sequences $(r_i)_{i \geq 0}$ and $(\rho_i)_{i \geq 0}$ such that:*

(i) *for all $i$, $\Gamma_i \vdash F_i$ is false in $M_i$ under the resource state $r_i$ and environment $\rho_i$;*

(ii) *if there is a trace $(\tau_i = P_i\mathbf{t_i})_{i \geq n}$ following some tail $(\Gamma_i \vdash F_i)_{i \geq n}$ of $(\Gamma_i \vdash F_i)_{i \geq 0}$, then there exists a sequence $(r'_i)_{i \geq n}$ of resource states such that $M, r'_i \models_{\rho_i} P_i\mathbf{t_i}$ for all $i \geq n$ and the sequence $(\alpha_i)_{i \geq n}$ of ordinals defined by $\alpha_i =$ least $\alpha$ s.t. $(r'_i, \rho_i(\mathbf{t_i})) \in P_i^\alpha$, is non-increasing and, furthermore, if $j$ is a progress point of $(\tau_i)$ then $\alpha_{j+1} < \alpha_j$.*

We remark that in order to prove this lemma, the resource states $r'_i$ used to

satisfy the trace formulas must be constructed as substates of the states $r_i$ used to falsify the sequents along the constructed path.

**Definition 4.7 ($\mathrm{CLBI}^{\omega}_{\mathrm{ID}}$ proof)** *A $CLBI^{\omega}_{ID}$ pre-proof $\mathcal{P} = (\mathcal{D}, \mathcal{R})$ is an* $\mathrm{CLBI}^{\omega}_{\mathrm{ID}}$ proof *if, for every infinite path in $\mathcal{D}$, there is an infinitely progressing trace following some tail of the path.*

Similar definitions, in different contexts, appear in [17,22].

**Proposition 4.8 (Soundness)** *If there is a $CLBI^{\omega}_{ID}$ proof of $\Gamma \vdash \Delta$ then $\Gamma \vdash \Delta$ is valid.*

**Proof.** (Sketch) Let $\mathcal{P}$ be a $\mathrm{CLBI}^{\omega}_{\mathrm{ID}}$ proof of $\Gamma \vdash F$. If $\Gamma \vdash F$ is not valid, i.e. false in some standard model $M$ in some resource state $r_0$ and environment $\rho_0$, then we can apply Lemma 4.6 to construct infinite sequences $(\Gamma_i \vdash F_i)_{i \geq 0}$, $(r_i)_{i \geq 0}$ and $(\rho_i)_{i \geq 0}$ satisfying the two properties of the lemma. As $(\Gamma_i \vdash \Delta_i)_{i \geq 0}$ is a path in $\mathcal{G}_{\mathcal{P}}$, there is an infinitely progressing trace following some tail of the path by Definition 4.7, so by the second property of the lemma we can construct an infinite descending chain of ordinals, which is a contradiction. □

**Example 4.9** The following is a $\mathrm{CLBI}^{\omega}_{\mathrm{ID}}$ proof of the sequent $F, Nx \vdash Nx$ (recall we gave an $\mathrm{LBI}_{\mathrm{ID}}$ proof in Example 3.8).

$$\dfrac{\dfrac{\dfrac{}{F \vdash N0}\,(NR_1)}{F, x = 0 \vdash Nx}\,(=\mathrm{L}) \qquad \dfrac{\dfrac{\dfrac{F, Nx \vdash Nx \;\; (\dagger)}{F, Ny \vdash Ny}\,(\mathrm{Subst})}{F, Ny \vdash Nsy}\,(NR_2)}{F, (x = sy; Ny) \vdash Nx}\,(=\mathrm{L})}{F, Nx \vdash Nx \;\; (\dagger)}\,(\mathrm{Case}\ N)$$

We use $(\dagger)$ to indicate the pairing of a suitable companion with the only bud in this pre-proof. To see that it is indeed a $\mathrm{CLBI}^{\omega}_{\mathrm{ID}}$ proof, observe that any infinite path $\pi$ in the pre-proof graph necessarily has a tail consisting of repetitions of the path from the companion to the bud in this pre-proof, and there is an progressing trace following this path: $(Nx, Ny, Ny, Ny, Nx)$ (with the progression occurring from the active formula $Nx$ of (Case $N$) to its case-descendant $Ny$ in the trace). Thus by concatenating copies of this trace we can obtain an infinitely progressing trace on a tail of $\pi$ as required.

We remark that, unlike the situation for $\mathrm{LBI}_{\mathrm{ID}}$ (cf. Example 3.8), we do not require generalisation in this proof, i.e., the invention of new formulas in the proof is not necessary.

**Example 4.10** The following is a $\mathrm{CLBI}^{\omega}_{\mathrm{ID}}$ pre-proof of the sequent $\mathtt{ls}\,x\,x' * \mathtt{ls}\,x'\,y \vdash \mathtt{ls}\,x\,y$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathtt{ls}\,x\,y \vdash \mathtt{ls}\,x\,y}\,(\mathrm{Id})}{I, \mathtt{ls}\,x\,y \vdash \mathtt{ls}\,x\,y}\,(\equiv)}{(x' = x; I), \mathtt{ls}\,x'\,y \vdash \mathtt{ls}\,x\,y}\,(=\mathrm{L}) \qquad \dfrac{\dfrac{\dfrac{}{x \mapsto z \vdash x \mapsto z}\,(\mathrm{Id}) \quad \dfrac{(\dagger)\;\; \mathtt{ls}\,x\,x', \mathtt{ls}\,x'\,y \vdash \mathtt{ls}\,x\,y}{\mathtt{ls}\,z\,x', \mathtt{ls}\,x'\,y \vdash \mathtt{ls}\,z\,y}\,(\mathrm{Subst})}{x \mapsto z, \mathtt{ls}\,z\,x', \mathtt{ls}\,x'\,y \vdash x \mapsto z * \mathtt{ls}\,z\,y}\,(*\mathrm{R})}{x \mapsto z, \mathtt{ls}\,z\,x', \mathtt{ls}\,x'\,y \vdash \mathtt{ls}\,x\,y}\,(\mathtt{ls}R_2)}{(\dagger)\;\; \mathtt{ls}\,x\,x', \mathtt{ls}\,x'\,y \vdash \mathtt{ls}\,x\,y}\,(\mathrm{Case}\ \mathtt{ls})}{\mathtt{ls}\,x\,x' * \mathtt{ls}\,x'\,y \vdash \mathtt{ls}\,x\,y}\,(*\mathrm{L})$$

The pairing of a suitable companion with the only bud in this pre-proof is denoted by (†). To see that the above is in fact a $\mathrm{CLBI}^{\omega}_{\mathrm{ID}}$ proof, observe that any infinite path $\pi$ in the pre-proof necessarily has a tail consisting solely of repetitions of the finite path in the pre-proof from the companion to the bud. Now, the sequence:

$$(\mathtt{ls}\,x\,x', \mathtt{ls}\,z\,x', \mathtt{ls}\,z\,x', \mathtt{ls}\,z\,x', \mathtt{ls}\,x\,x')$$

is a trace following this path. It progresses at its first position because $\mathtt{ls}\,z\,x'$ is a case-descendant of $\mathtt{ls}\,x\,x'$ in the displayed application of the case-split rule (Case $\mathtt{ls}$). Thus one can readily see that one can build the required infinitely progressing trace on this tail of $\pi$ by concatenating copies of this trace, and so this pre-proof is indeed a $\mathrm{CLBI}^{\omega}_{\mathrm{ID}}$ proof.

We remark that the $\mathrm{LBI}_{\mathrm{ID}}$ proof of $\mathtt{ls}\,x\,x' * \mathtt{ls}\,x'\,y \vdash \mathtt{ls}\,x\,y$ proceeds, after applying ($*$L), by induction on $\mathtt{ls}\,x\,x'$ using the induction variables $z, z'$ (say) and the induction hypothesis $\mathtt{ls}\,z'\,y \mathrel{-\!\!*} \mathtt{ls}\,z\,y$, thus requiring a generalisation similar to that needed in Example 3.8.

**Proposition 4.11** *It is decidable whether a $CLBI^{\omega}_{ID}$ pre-proof is a $CLBI^{\omega}_{ID}$ proof.*

**Proof.** (Sketch) The property of every infinite path posessing an infinitely progressing trace along a tail is an $\omega$-regular property, and hence reducible to the emptiness of a Büchi automaton. A full proof (for a general notion of trace) appears in [9]; a similar argument appears in [22]. □

# 5 Conclusions and Future work

Our work thus far constitutes a reasonably straightforward extension, to the setting of BI, of the inductive definition framework and proof systems formulated in our previous work for first-order logic with inductively defined relations [8,10,9]. Thus one might reasonably hope that the key proof-theoretical results from that work, including appropriate completeness and cut-elimination theorems, will also extend to the systems we consider here. One would also expect to be able to show that our system $\mathrm{CLBI}^{\omega}_{\mathrm{ID}}$ subsumes $\mathrm{LBI}_{\mathrm{ID}}$, with the question of their equivalence presenting similar difficulties to those discussed in [8,10,9]; for first-order logic with inductively defined relations, it is not yet known how to extract a traditional induction proof from a cyclic proof, and we have only conjectured the equivalence of the two proof styles in this setting.

One could also examine the extension of our inductive definition framework to more powerful induction schemas, for example definitions employing additive and/or multiplicative implication, and to coinduction.

For the immediate future, though, we intend to pursue two main avenues. First, we would like to further explore the potential of cyclic proof for proof search in $\mathrm{BI}_{\mathrm{ID}}$ and related sublogics (e.g. separation logic with inductive definitions). We have already seen trivial examples in which cyclic proof appears to avoid the generalisation necessary in the corresponding inductive proof (Examples 4.9 and 4.10). More generally, cyclic proof should offer a "least-commitment" approach to proof search, whereby the induction schema, variables and hypotheses are not chosen at the beginning of the proof, as in traditional inductive theorem proving, but are eventually selected implicitly via the satisfaction of the soundness condition. We

have already given proof-theoretic machinery for analysing and manipulating the structure of general cyclic proofs [9] which may be of assistance in such investigations. Secondly, we hope that it will be possible to directly formulate cyclic proof systems for the direct verification of low-level programs, (using, e.g., Hoare triples). We speculate that such systems would use appropriate cyclic proof principles to establish invariants for the looping constructs in such programs, or, given such invariants, to prove appropriate postconditions.

# References


[1] Peter Aczel. An introduction to inductive definitions. In Jon Barwise, editor, *Handbook of Mathematical Logic*, pages 739–782. North-Holland, 1977.

[2] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *Proceedings of FMCO 2005*, volume 4111 of *LNCS*, pages 115–137, 2005.

[3] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Symbolic execution with separation logic. In *Proceedings of APLAS 2005*, volume 3780 of *LNCS*, pages 52–68. Springer, 2005.

[4] Josh Berdine, Byron Cook, Dino Distefano, and Peter W. O'Hearn. Automatic termination proofs for programs with shape-shifting heaps. In *Proceedings of 18th CAV*, volume 4144 of *LNCS*, pages 386–400. Springer, 2006.

[5] Bodil Biering, Lars Birkedal, and Noah Torp-Smith. BI hyperdoctrines and separation logic. In *Proceedings of ESOP'05*, pages 233–247. Springer-Verlag, 2005.

[6] L. Birkedal, N. Torp-Smith, and J.C. Reynolds. Local reasoning about a copying garbage collector. In *Proceedings of POPL'04*, pages 220–231, 2004.

[7] Richard Bornat, Cristiano Calcagno, and Peter O'Hearn. Local reasoning, separation and aliasing. In *Proceedings of SPACE'04*, January 2004.

[8] James Brotherston. Cyclic proofs for first-order logic with inductive definitions. In B. Beckert, editor, *Automated Reasoning with Analytic Tableaux and Related Methods: Proceedings of TABLEAUX 2005*, volume 3702 of *LNAI*, pages 78–92. Springer-Verlag, 2005.

[9] James Brotherston. *Sequent Calculus Proof Systems for Inductive Definitions*. PhD thesis, University of Edinburgh, November 2006.

[10] James Brotherston and Alex Simpson. Complete sequent calculi for induction and infinite descent. In preparation, 2007.

[11] Alan Bundy. The automation of proof by mathematical induction. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 13, pages 845–911. Elsevier Science, 2001.

[12] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Beyond reachability: Shape abstraction in the presence of pointer arithmetic. In *Proceedings of SAS-13*, volume 4134 of *LNCS*, pages 182–203. Springer, 2006.

[13] Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. A local shape analysis based on separation logic. In *Proceedings of TACAS-12*, volume 3920 of *LNCS*, pages 287–302, 2006.

[14] Samin Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In *Proceedings of POPL'01*, January 2001.

[15] Per Martin-Löf. Haupstatz for the intuitionistic theory of iterated inductive definitions. In J.E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, pages 179–216. North-Holland, 1971.

[16] Yiannis N. Moschovakis. *Elementary Induction on Abstract Structures*, volume 77 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1974.

[17] Damian Niwiński and Igor Walukiewicz. Games for the $\mu$-calculus. *Theoretical Computer Science*, 163:99–116, 1997.

[18] P.W. O'Hearn and D. J. Pym. The logic of bunched implications. *Bulletin of Symbolic Logic*, 5(2):215–244, June 1999.

[19] David Pym. *The Semantics and Proof Theory of the Logic of Bunched Implications*. Applied Logic Series. Kluwer, 2002.

[20] Greg Restall. *An Introduction to Substructural Logics*. Routledge, 2000.

[21] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, 2002.

[22] Christoph Sprenger and Mads Dam. A note on global induction mechanisms in a $\mu$-calculus with explicit approximations. *Theoretical Informatics and Applications*, July 2003. Full version of FICS '02 paper.

[23] Hongseok Yang. An example of local reasoning in BI pointer logic: the Schorr-Waite graph marking algorithm. In *Proceedings of SPACE 2001*, 2001.

# Verifying Complex Properties using Symbolic Shape Analysis

Thomas Wies

*Departement of Computer Science*
*University of Freiburg, Germany*

Viktor Kuncak Karen Zee Martin Rinard

*MIT, CSAIL*
*Cambridge, USA*

Andreas Podelski

*Departement of Computer Science*
*University of Freiburg, Germany*

**Abstract**

One of the main challenges in the verification of software systems is the analysis of statically unbounded data structures with dynamic memory allocation, such as linked data structures and arrays. We describe Bohne, a new analysis for verifying data structures. Bohne verifies data structure operations and shows that 1) the operations preserve data structure invariants and 2) the operations satisfy their specifications expressed in terms of changes to the set of objects stored in the data structure. During the analysis, Bohne infers loop invariants in the form of disjunctions of universally quantified Boolean combinations of formulas, represented as sets of binary decision diagrams. To synthesize loop invariants of this form, Bohne uses a combination of decision procedures for Monadic Second-Order Logic over trees, SMT-LIB decision procedures (currently CVC Lite), first-order provers such as SPASS and E, and the automated reasoner within the Isabelle interactive theorem prover. This architecture shows that synthesized loop invariants can serve as a useful communication mechanism between different decision procedures. In addition, Bohne uses field constraint analysis, a combination mechanism that enables the use of uninterpreted function symbols within formulas of Monadic Second-Order Logic over trees. Using Bohne, we have verified operations on data structures such as linked lists with iterators and back pointers, trees with and without parent pointers, two-level skip lists, array data structures, and sorted lists. We have deployed Bohne in the Hob and Jahob data structure analysis systems, enabling us to combine Bohne with analyses of data structure clients and apply it in the context of larger programs. This paper describes the Bohne algorithm, the techniques that Bohne uses to reduce the amount of annotations and the running time of the analysis.

## 1 Introduction

Complex data structure invariants are one of the main challenges in verifying software systems. Unbounded data structures such as linked data structures and dynamically allocated arrays make the state space of software artifacts infinite and require new reasoning techniques (such as reasoning about reachability) that have traditionally not been part of theorem provers specialized for program verification. The ability of linked structures to change their shape makes them a powerful programming construct, but at the same time makes them difficult to analyze, because the appropriate analysis representation is dependent on the invariants that the program maintains. It is therefore not surprising that the most successful verification approaches for analysis of data structures use parameterized abstract domains; these analyses include parametric shape analysis [47] as well as predicate abstraction [3,23] and its generalizations [14, 31].

This paper presents *Bohne*, an algorithm for inferring loop invariants of programs that manipulate heap-allocated data structures. Like predicate abstraction, Bohne is parameterized by the properties to be verified. What makes the Bohne algorithm unique is the use of a precise abstraction domain that can express detailed properties

of different regions of a program's infinite memory, and a range of techniques for exploring this analysis domain using decision procedures. The algorithm was initially developed as a symbolic shape analysis [53,43] for linked data structures and uses the key idea of many shape analyses, made explicit in the TVLA analyzer [47,36]: the partitioning of objects according to certain unary predicates. One of the observations of our paper is that the synthesis of heap partitions is not only useful for analyzing shape properties (which involve transitive closure), but also for combining such shape properties with sorting properties of data structures and properties expressible using linear arithmetic and first-order logic.

### *1.1 Related Work*

We next put the Bohne algorithm in the context of two abstract interpretation [11] approaches that are closest to symbolic shape analysis: predicate abstraction and parametric shape analysis. We then discuss the work on decision procedures because Bohne uses a validity checker for an expressive logic to perform the analysis.

**Predicate abstraction.** Bohne builds on predicate abstraction but introduces important new techniques that make it applicable to the domain of shape analysis. There are two main sources of complexity of loop invariants in shape analysis. The first source of complexity is the fact that the invariants contain reachability predicates. To address this problem, Bohne uses a decision procedure for monadic second-order logic over trees [50, 25], and combines it with uninterpreted function symbols in a way that preserves completeness in important cases [54]. The second source of complexity is that the invariants contain universal quantifiers in an essential way. Among the main approaches for dealing with quantified invariants in predicate abstraction is the use of Skolem constants [14], indexed predicates [31] and the use of abstraction predicates that contain quantifiers. The key difficulty in using Skolem constants for shape analysis is that the properties of individual objects depend on the "context", given by the properties of surrounding objects, which means that it is not enough to use a fixed Skolem constant throughout the analysis; it is instead necessary to instantiate universal quantifiers from previous loop iterations, in some cases multiple times. Compared to indexed predicates [31] the domain used by Bohne is more general because it contains disjunctions of universally quantified statements. The presence of disjunctions is not only more expressive in principle, but allows Bohne to keep formulas under the universal quantifiers more specific. This enables the use of less precise, but more efficient algorithms for computing changes to properties of objects without losing too much precision in the overall analysis. Finally, the advantage of using abstraction tailored to shape analysis compared to using quantified global predicates is that the parameters to shape-analysis-oriented abstraction are properties of objects in a state, as opposed to global properties of a state, and the number of global predicates needed to emulate state predicates is exponential in the number of properties [39, 53].

The advantages of combining predicate abstraction with shape analysis are clearly demonstrated in lazy shape analysis [6]. Lazy shape analysis performs independent runs of a shape analysis algorithm, whose results are then used to improve the precision of predicate abstraction. In contrast, our symbolic shape analysis gen-

eralizes predicate abstraction technique to the point where it itself becomes effective as a shape analysis. We note that Bohne has also been extended to perform the automated discovery of predicates; the discussion of this extension is beyond the scope of this paper.

**Shape analysis.** Shape analyses are precise analyses for linked data structures. They were originally used for compiler optimizations [24, 19, 18] and lacked the precision needed to establish invariants that Bohne is analyzing. Precise data structure analyses for verification include [29, 16, 26, 35, 41, 20, 47] and have recently also been applied to verify set implementations [45]. Unlike Bohne, most shape analyses that synthesize loop invariants are based on precomputed transfer functions and a fixed (though parameterized) set of properties to be tracked; recent approaches enable automation of such computation using decision procedures [57, 56, 58, 43, 54] or finite differencing [46]. Our approach differs from [32] in using complete reasoning about reachability in both lists and trees, and using a different architecture of the reasoning procedure. Our reasoning procedure uses a coarse-grain combination of reachability reasoning with decision procedures and theorem provers for numerical and first-order properties, as opposed to using a Nelson-Oppen style theorem prover. This allowed us to easily combine several tools that were developed completely independently [25, 4, 42]. Shape analysis approaches have also been used to verify sortedness properties [38] relying on manually abstracting a sortedness relation.

**Decision procedures.** Our symbolic shape analysis algorithm relies on decision procedures for expressive logics to perform synthesis of loop invariants. The system then verifies that the synthesized invariants are sufficient to prove the absence of errors and to prove the postcondition. During the invariant synthesis, the analysis primarily uses the MONA decision procedure [25] with field constraint analysis [54] to reason about expressive invariants involving reachability in tree-like linked structures. Our analysis also uses CVC Lite [4] via the SMT-LIB interface to reason about array data structures, local properties in non-tree data structures, and linear arithmetic. These two decision procedures are most relevant for the present paper. In our system (Figure 1) we also use interactive theorem provers Isabelle [42] and Coq [5] to debug the proof obligations and translations into decision procedures, as well as to automatically discharge some proof obligations using simplication and proof search built into these provers. We have also had success using first-order theorem provers Vampire [51], E [48] (via the TPTP interface [49]) as well as SPASS [52]. We used first-order theorem provers in Jahob to verify implementations of data structures [8], avoiding the use of reachability using specification variables similarly to the approaches taken in [29, 40] and automated to some extent in [37]. Finally, to reason about the sizes of data structures, we used a new decision procedure [30, 28] with a reduction to Presburger arithmetic.

Several recent decision procedures address specifically linked lists [1, 12, 39, 7, 44], where the emphasis is on the predictability (decision procedures for well-defined classes of properties of linked lists), the efficiency (membership in NP), the ability to interoperate with other reasoning procedures, and modularity. Although the Bohne approach is not limited to lists, it can take advantage of decision procedures for lists by applying such specialized procedures when they are applicable and using

more general reasoning otherwise. In our current experience, the MONA decision procedure [25] proved to be effective for verifying reachability in both list and tree structures.

Bohne could also take advantage of logics for reasoning about reachability, such as the logic of reachable shapes [55]. Existing logics, such as guarded fixpoint logic [21] and description logics with reachability [10,17] are attractive because of their expressive power, but so far no decision procedures for these logics have been implemented.

### *1.2 Contributions*

We have previously described the general idea of symbolic shape analysis [43] as well as the field constraint analysis decision procedure for combining reachability reasoning with uninterpreted function symbols [54]. These previous techniques are our starting point. The main contributions of this paper are the following:

(i) We describe a method for synthesis of Boolean heap programs that improves the efficiency of fixpoint evaluation by precomputing abstract transition relations and can control the precision/efficiency trade-off by recomputing transition relations on-demand during fixpoint computation.

(ii) We introduce semantic caching of decision procedure queries across different fixpoint iterations and even different analyzed procedures. The caching yields substantial improvements for procedures that exhibit some similarity, which opens up the possibility of using our analysis in an interactive context.

(iii) We describe a static analysis that propagates precondition conjuncts and quickly finds many true facts, reducing the running time and the number of needed abstraction predicates for the subsequent symbolic shape analysis.

(iv) We present a domain-specific quantifier instantiation technique that significantly improves the running time of the analysis. Furthermore, it often eliminates the need for the underlying decision procedures to deal with quantifiers.

Together, these new techniques allowed us to verify a range of data structures without specifying loop invariants and without specifying a large number of abstraction predicates. Our examples include implementations of lists (with iterators and with back pointers), trees with parent pointers, two-level skip lists, sorted lists, as well as combinations of these data structures. What makes these results particularly interesting is a higher level of automation than in previous approaches: Bohne synthesizes loop invariants that involve reachability expressions and numerical quantities, yet it does not have precomputed transfer functions for a particular set of abstraction predicates. Bohne instead uses decision procedures to reason about arbitrary predicates definable in a given logic. Moreover, in our system the developer is not required to manually specify the changes of membership of elements in sets because such changes are computed by the system. Our system uses such synthesized invariants to communicate the information between different decision procedures.

**Bohne as a component of Jahob.** Bohne is part of the data structure verification frameworks Jahob [27, 28] and Hob [34, 33]. The goal of these systems is to verify data structure consistency properties in the context of non-trivial programs. To achieve this goal, these tools combine multiple static analyses, theorem proving, and

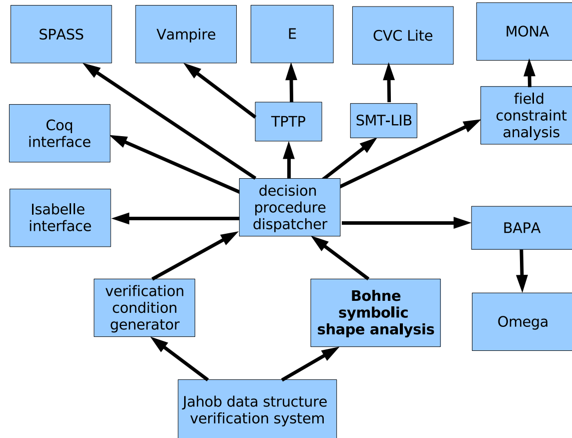



Fig. 1. Jahob Data Structure Analysis System Architecture

decision procedures. In this paper we present our experience in deploying Bohne in the Jahob framework. The input language for Jahob is a subset of Java extended with annotations written as special comments. Therefore, Jahob programs can be compiled and executed using existing Java compilers and virtual machines.

Figure 1 illustrates the integration of Bohne into the Jahob framework. Bohne uses Jahob's facilities for symbolic execution of program statements and the validity checker to compute the abstraction of the source program. The output of Bohne is the source program annotated with the inferred loop invariants. The annotated program serves as an input to a verification condition generator. The generated verification conditions are verified using an approach [28] that combines special purpose decision procedures, general purpose theorem provers, and reasoning techniques such as field constraint analysis [54].

## 2 Motivating Example

We illustrate our technique on the procedure `SortedList.insert` shown in Figure 2. This procedure inserts a `Node` object into a global sorted list. The annotation given by special comments `/*: ... */` consists of data structure invariants, pre- and postconditions, as well as hints for the analysis. Formulas are expressed in a subset of the language used in the Isabelle interactive theorem prover [42]. The specification uses an abstract set variable `content` which is defined as the set of non-null objects reachable from the global variable `first` by following the field `next`. The data structure invariants are specified by the annotation `invariant "..."`. For instance, the first invariant expresses the fact that the field `next` forms trees in the heap, i.e. that `next` is acyclic and injective; the third invariant expresses the fact that the elements stored in the list are sorted in increasing order according to field `data`. The precondition of the procedure, `requires "..."`, states that the object to be inserted is non-null and not yet contained in the list. The postcondition, `ensures "..."`, expresses that the argument is properly inserted into the list.

The loop in the procedure body traverses the list until it finds the proper position for insertion. It then inserts the argument such that the resulting data structure is again a sorted list. Our analysis, Bohne, is capable of verifying that the postcondition holds at the end of the procedure `insert`, that data structure invariants are

```
class Node {
  public int data;
  public Node next;
}
class SortedList {
  private static Node first;
  /*: public static specvar content :: objset;
      vardefs "content == {v. v ≠ null ∧ next* first v}";

      invariant "tree [next]";
      invariant "first = null ∨ (∀ n. n.next ≠ first)";
      invariant "∀ v. v ∈ content ∧ v.next ≠ null ⟶ v..Node.data ≤ v.next.data";
      invariant "∀ v. v ≠ null ∧ v.next ≠ null ⟶ v.next ∈ content";
  */
  public static void insert(Node n)
    /*: requires "n ≠ null ∧ n ∉ content"
        modifies content
        ensures "content = old content ∪ {n}" */
  {/*: specvar reach_curr :: objset;
       vardefs "reach_curr == {v. next* curr v}";
       specvar prev_le_n :: bool;
       vardefs "prev.data <= n.data"; */
    Node prev = null;
    Node curr = first;
    while ((curr != null) && (curr.data < n.data)) {
      //: track(reach_curr); track(prev_le_n);
      prev = curr;
      curr = curr.next;
    }
    n.next = curr;
    if (prev != null) prev.next = n;
    else first = n;
  }
}
```

Fig. 2. Insertion into a sorted list

```
tree [next] ∧ (first = null ∨ (∀ n. n.next ≠ first)) ∧
(∀ v. v ∈ content ∧ v.next ≠ null ⟶ v.data <= v.next.data) ∧
(∀ v w. v ≠ null & w ≠ null & v.next = w ⟶ w ∈ content) ∧
n ∧ null ∧ n ∉ content ∧ content = old content ∧
(curr ∧ null ⟶ curr ∈ content) ∧ (prev = null ⟶ first = curr) ∧
(prev ∧ null ⟶ prev ∈ content ∧ prev ∉ reach_curr ∧
                       prev.next = curr ∧ prev_le_n)
```

Fig. 3. Loop invariant for procedure `SortedList.insert`

preserved, and that there are no run-time errors such as null pointer dereferences. In order to establish these properties, Bohne derives a complex loop invariant shown in Fig. 3.

The main difficulties for inferring this invariant are: (1) it contains universal quantifiers over an unbounded domain and (2) it contains constructs such as reachability, numerical properties, and uninterpreted function symbols.

Bohne infers universally quantified invariants using symbolic shape analysis based on Boolean heaps [53, 43]. This approach can be viewed as a generalization of predicate abstraction or a symbolic approach to parametric shape analysis. Abstraction predicates can be Boolean-valued state predicates (which are either true or false in a given state, such as `prev_le_n`) or predicates denoting sets of heap objects in a given state (which are true of a *given object* in a *given state*, such as `reach_curr`). The latter serve as building blocks of the inferred universally quantified invariants. The `track(...)` annotation is used as a hint on which predicates

$$
\begin{array}{l}
\text{I} \wedge \neg(\text{curr.data} < \text{n.data}) \wedge \text{prev} \neq \text{null} \wedge \\
\text{next_1} = \text{next}[\text{n} := \text{curr}][\text{prev} := \text{n}] \wedge \\
\text{content_1} = \{\text{v. v} \neq \text{null} \wedge \text{next_1}^* \text{ first v}\} \wedge \\
\text{v} \in \text{content_1} \wedge \text{v.next_1} \neq \text{null} \longrightarrow \text{v.data} \leq \text{v.next_1.data}
\end{array}
$$

Fig. 4. Verification condition for preservation of sortedness

the analysis should use for the abstraction of which code fragments.

To reduce the annotation burden we use a syntactic analysis to infer abstraction predicates automatically. Furthermore, parts of the invariant often literally come from the procedure's precondition. In particular, data structure invariants are often preserved as long as the heap is not mutated. We therefore precede the symbolic shape analysis phase with an analysis that propagates precondition conjuncts across the control-flow graph of the procedure's body. Using this propagation technique we are able to infer the first six conjuncts of the invariant. The symbolic shape analysis phase makes use of this partial invariant to infer the full invariant shown in Fig. 3.

Bohne's symbolic shape analysis enables the combination of different decision procedures. Thereby the inferred invariants communicate information between the individual decision procedures, as illustrated with the following example. Figure 4 shows one of the generated verification conditions for the `insert` procedure. It expresses the fact that the sortedness property is reestablished after executing the path from the exit point of the loop through the if-branch of the conditional to the procedure's return point. The symbol "`I`" denotes the loop invariant given in Fig. 3. This verification condition is valid. Its proof requires the fact `content'` $=$ `content` $\cup$ $\{$`n`$\}$; denote this fact $P$. $P$ follows from the given assumptions. The MONA decision procedure is able to conclude $P$ by expanding the definitions of the abstract sets `content` and `content'`. However, MONA is not able to prove the verification condition, because proving its conclusion requires reasoning over integers. On the other hand, the CVC Lite decision procedure is able to prove the conclusion given the fact $P$ by reasoning over the abstract sets without expanding their definitions, but it is not able to conclude $P$ from the assumptions, because this deduction step requires reasoning over reachability. In order to communicate $P$ between the two decision procedures, Bohne infers, in addition to the loop invariant `I`, an invariant for the procedure's return point that includes the missing fact $P$. This invariant enables CVC Lite to prove the verification condition.

## 3 Context-Sensitive Abstraction

We next describe the symbolic shape analysis algorithm implemented in Bohne. What makes this algorithm unique is the fact that abstract transition relations are computed on-demand in each fixpoint iteration taking into account the *context* that approximates previously explored abstract states. This approach allows the algorithm to take advantage of precomputed abstract transition relations from previous fixpoint iterations, while maintaining sufficient precision for the analysis of linked data structures by recomputing the transitions when the context changes in a significant way.

```
proc Reach(init : precondition formula,
           ℓ_init : initial program location,
           T : set of guarded commands) =
  let init# = abstract(init)
  let root = ⟨location = ℓ_init; states = init#; sons = ∅⟩
  let unprocessed = {root}
  while unprocessed ≠ ∅ do
    choose n ∈ unprocessed
    for all (n.location, c, ℓ′) ∈ T do
      let context = { m.states | m.location = n.location }
      let old = { m.states | m.location = ℓ′ }
      let new = AbstractPost(c, context, n.states) − old
      if new ≠ ∅ then
        let n′ = ⟨location = ℓ′; states = new; sons = ∅⟩
        n.sons := n.sons ∪ {(c, n′)}
        unprocessed := unprocessed ∪ {n′}
    unprocessed := unprocessed − {n}
  return root
```

Fig. 5. Reachability analysis in Bohne

### *3.1 Reachability Analysis*

The input of Bohne is the procedure to be analyzed, preconditions specifying the initial states of the procedure, and a set of abstraction predicates. Bohne converts the procedure into a set of guarded commands that correspond to the loop-free paths in the control-flow graph.

The pseudo code of Bohne's top-level fixpoint computation loop is shown in Figure 5. The analysis first abstracts the conjunction of the procedure's preconditions obtaining an initial set of abstract states. It then computes an abstract reachability tree. Each node in this tree is labeled by a program location and a set of abstract states, the root being labeled by the initial location and the abstraction of the preconditions. The edges in the tree are labeled by guarded commands. The reachability tree keeps track of abstract traces which are used for the analysis of abstract counterexamples.

For each unprocessed node in the tree, the analysis computes the abstract postcondition for the associated abstract states and all outgoing transitions of the corresponding program location. Transitions are abstracted context-sensitively, taking into account the previously discovered reachable abstract states for the associated program location. Whenever the difference between the already discovered abstract states of the post location and the abstract post states of the processed transition is non-empty, a new unprocessed node is added to the tree. The analysis stops after the list of unprocessed nodes becomes empty, indicating that the fixpoint is reached. After termination of the reachability analysis, Bohne annotates the original procedure with the computed loop invariants and passes the result to the verification condition generator.

Focusing on algorithmic aspects, we next give a description of the abstract do-

main, abstraction function, and the abstract post operator.

### 3.2 Symbolic Shape Analysis

Following the framework of abstract interpretation [11], a static analysis is defined by lattice-theoretic domains and by fixpoint iteration over the domains. Symbolic shape analysis can be seen as a generalization of predicate abstraction [22]. For *predicate abstraction* the analysis computes an invariant; the fixpoint operator is an abstraction of the *post* operator; the concrete domain consists of sets of states (represented by closed formulas), and the abstract domain of a finite lattice of closed formulas.

**Abstract Domain.** Let $\mathsf{Pred}$ be a finite set of abstraction predicates $p(v)$ with an implicit free variable $v$ ranging over heap objects. A *cube* $C$ is a partial mapping from $\mathsf{Pred}$ to $\{0, 1\}$. We call a total cube *complete*. We say that predicate $p$ occurs positively (occurs negatively, does not occur) in $C$ if $C(p) = 1$ ($C(p) = 0$, $C(p)$ is undefined). We denote by $\mathsf{Cubes}$ the set of all cubes. An abstract state is a subset of cubes, which we call a *Boolean heap*. The abstract domain is given by sets of Boolean heaps, i.e. sets of sets of cubes: $\mathsf{AbsDom} = 2^{2^{\mathsf{Cubes}}}$.

**Meaning Function.** The meaning function $\gamma$ is defined on cubes, Boolean heaps, and sets of Boolean heaps as follows:

$$\gamma(C) = \bigwedge_{p \in \mathsf{Pred} \cap \mathsf{dom}(C)} p^{C(p)}, \quad \gamma(H) = \forall v. \bigvee_{C \in H} \gamma(C), \quad \gamma(\mathcal{H}) = \bigvee_{H \in \mathcal{H}} \gamma(H)$$
$$\text{where } p^1 = p \text{ and } p^0 = \neg p$$

The meaning of a cube $C$ is the conjunction of the properly signed predicates in $\mathsf{Pred}$. A Boolean heap $H$ describes all concrete states whose heap is partitioned according to the cubes in $H$. The meaning of a set $\mathcal{H}$ of Boolean heaps is the disjunction of the meaning of all its elements.

**Lattice Structure.** Define a partial order $\sqsubseteq$ on cubes by:

$$C \sqsubseteq C' \stackrel{def}{\iff} \forall p \in \mathsf{Pred}.\ C'(p) = C(p) \ \lor \ (C'(p) \text{ is undefined}).$$

For a cube $C$ and Boolean heap $H$ we write $C \in_c H$ as a short notation for the fact that $C$ is complete and there exists $C' \in H$ such that $C \sqsubseteq C'$. The partial order $\sqsubseteq$ is extended from cubes to a preorder on Boolean heaps:

$$H \sqsubseteq H' \stackrel{def}{\iff} \forall C \in H.\, \exists C' \in H'.\, C \sqsubseteq C'.$$

For notational convenience we identify Boolean heaps up to subsumption of cubes, i.e. up to equivalence under the relation ($\sqsubseteq \cap \sqsubseteq^{-1}$). We then identify $\sqsubseteq$ with the partial order on the corresponding quotient of Boolean heaps. In the same way we extend $\sqsubseteq$ from Boolean heaps to a partial order on the abstract domain. These partial orders induce Boolean algebra structures. We denote by $\sqcap$, $\sqcup$ and $\overline{\cdot}$ the meet, join and complement operations of these Boolean algebras. Bohne usees binary decision diagrams (BDDs) [9] to implement Boolean heaps, the abstract domain, and operations of the Boolean algebras.

**Context-sensitive Cartesian post.** The abstract post operator implemented in Bohne is a refinement of the abstract post operator presented in [43]. Its core is given by the *Cartesian post operator*. This operator maps a guarded command $c$, and a set of Boolean heaps $\mathcal{H}$ to a set of Boolean heaps as follows:

$$\begin{array}{l}\mathsf{CartesianPost}(c, \mathcal{H}) = \\ \quad \textbf{let } \mathsf{cpost}(c, C) = \bigsqcap \{\, C' \mid \forall p \in \mathsf{Pred}.\ C \sqsubseteq \mathsf{wlp}^{\#}(c, p^{C'(p)})\,\} \\ \quad \textbf{in } \{\, \{\, \mathsf{cpost}(c, C) \mid C \in_c H \,\} \mid H \in \mathcal{H} \,\}.\end{array}$$

The actual abstraction occurs in the computation of $\mathsf{wlp}^{\#}(c, F)$ which is defined by:

$$\mathsf{wlp}^{\#}(c, F) = \{\, C \mid \gamma(C) \models \mathsf{wlp}(c, F)\,\} \ .$$

The Cartesian post maps each Boolean heap $H$ in $\mathcal{H}$ to a new Boolean heap $H'$. For a given state $s$ satisfying $\gamma(H)$, a cube $C$ in $H$ represents a set of heap objects in $s$. The Cartesian post computes the local effect of command $c$ on each set of objects which is represented by some complete cube in $H$: each complete cube $C$ in $H$ is mapped to the smallest cube $\mathsf{cpost}(c, C)$ that represents at least the same set of objects in the post states under command $c$. Consequently, each object in a given post state is represented by some cube in the resulting Boolean heap $H'$, i.e. all post states satisfy $\gamma(H')$. The effect of $c$ on the objects represented by some cube is expressed in terms of weakest preconditions $\mathsf{wlp}$ of abstraction predicates. These are abstracted by $\mathsf{wlp}^{\#}$.

Computing the effect of $c$ for each cube in $H$ locally implies that we do not take into account the full information provided by $H$. This becomes an inherent problem if updated predicates express non-local properties such as reachability. As an example, consider a Boolean heap $H$ that contains two cubes

$$\begin{array}{l}C_1 = [(x = v) \mapsto 1, (y = v) \mapsto 0, (\mathsf{next}^* z\ v) \mapsto 1] \text{ and} \\ C_2 = [(x = v) \mapsto 0, (y = v) \mapsto 1, (\mathsf{next}^* z\ v) \mapsto 0] \ .\end{array}$$

Cube $C_1$ describes an object which is pointed to by a stack variable $x$ and reachable from some other stack variable $z$ following field $\mathsf{next}$. Cube $C_2$ describes a second object which is pointed to by stack variable $y$, but which is not reachable from $z$. If we consider a field update ($c = (x.\mathsf{next} := y)$) then after the update $y$ is reachable from $z$. However we have

$$\mathsf{cpost}(c, C_2) \not\sqsubseteq [(\mathsf{next}^* z\ v) \mapsto 1]$$

because $C_2$ is updated independently of $C_1$. In principle one can strengthen the abstraction of weakest preconditions by taking into account the Boolean heap $H$ for which the post is computed. In fact we have

$$\gamma(H) \wedge (y = v) \models \mathsf{wlp}(c, \mathsf{next}^* z\ v) \ .$$

This strengthening would result in a more precise Cartesian post, but as a consequence abstract weakest preconditions would have to be recomputed for each Boolean heap to which the Cartesian post is applied. This would make the analysis

```
proc CSCartesianPost(c, Γ : context formula, H : AbsDom) : AbsDom =
  let c# = Cubes
  if c# is precomputed for (c,Γ) then c# := lookup(c,Γ)
  else foreach p ∈ Pred do
    c# := c# ⊓ ( [p' ↦ 1] ⊓ ¬wlp#(c,Γ,¬p) ⊔
                 [p' ↦ 0] ⊓ ¬wlp#(c,Γ,p) )
  let H' = ∅
  foreach H ∈ H do
    let H' = RelationalProduct(H, c#)
    H' := H' ⊔ {H'}
  return H'
```

$$
\begin{array}{l}
\textbf{proc}\ \mathsf{CSCartesianPost}(c,\ \Gamma : \text{context formula},\ \mathcal{H} : \mathsf{AbsDom}) : \mathsf{AbsDom} = \\
\quad \textbf{let}\ \mathsf{c}^{\#} = \mathsf{Cubes} \\
\quad \textbf{if}\ c^{\#}\ \text{is precomputed for}\ (c,\Gamma)\ \textbf{then}\ c^{\#} := \text{lookup}(c,\Gamma) \\
\quad \textbf{else foreach}\ p \in \mathsf{Pred}\ \textbf{do} \\
\qquad c^{\#} := c^{\#} \sqcap \left( \begin{array}{l} [p' \mapsto 1] \sqcap \overline{\mathsf{wlp}^{\#}(c,\Gamma,\neg p)} \sqcup \\ {[p' \mapsto 0]} \sqcap \overline{\mathsf{wlp}^{\#}(c,\Gamma,p)} \end{array} \right) \\
\quad \textbf{let}\ \mathcal{H}' = \emptyset \\
\quad \textbf{foreach}\ H \in \mathcal{H}\ \textbf{do} \\
\qquad \textbf{let}\ H' = \mathsf{RelationalProduct}(H, c^{\#}) \\
\qquad \mathcal{H}' := \mathcal{H}' \sqcup \{H'\} \\
\quad \textbf{return}\ \mathcal{H}'
\end{array}
$$

Fig. 6. Context-sensitive Cartesian post

infeasible. Nevertheless, such global context information is valuable when updated predicates describe global properties such as reachability. Therefore, we would like to strengthen the abstraction using some global information, accepting that abstract weakest preconditions have to be recomputed occasionally. We introduce the *context-sensitive Cartesian post* to allow this kind of strengthening:

$$
\begin{array}{ll}
 & \mathsf{CSCartesianPost}(c,\Gamma,\mathcal{H}) = \\
 & \textbf{let}\ \mathsf{cpost}(c,C) = \bigsqcap\{\, C' \mid \forall p \in \mathsf{Pred}.\ C \sqsubseteq \mathsf{wlp}^{\#}(c,\Gamma,p^{C'(p)})\,\} \\
 & \textbf{in}\ \{\,\{\,\mathsf{cpost}(c,C) \mid C \in_c H\,\} \mid H \in \mathcal{H}\,\} \\
\text{where} & \mathsf{wlp}^{\#}(c,\Gamma,F) = \{\, C \mid \Gamma \wedge \gamma(C) \models \mathsf{wlp}(c,F)\,\}
\end{array} \tag{1}
$$

The formula $\Gamma$ is the key tuning parameter that controls the tradeoff between precision and efficiency of the analysis. We impose a restriction on $\Gamma$: we say that $\Gamma$ is a *context formula* for a set of Boolean heaps $\mathcal{H}$ if $\gamma(\mathcal{H})$ implies $\Gamma$. In order to ensure soundness, we require that for all applications $\mathsf{CSCartesianPost}(c,\Gamma,\mathcal{H})$ of the context-sensitive Cartesian post $\Gamma$ is a context formula for $\mathcal{H}$.

Figure 6 gives an implementation of the context-sensitive Cartesian post operator that exploits the representation of Boolean heaps as BDDs. First it precomputes an abstract transition relation $c^{\#}$ which is expressed in terms of cubes over primed and unprimed abstraction predicates. After that it computes the relational product of $c^{\#}$ and each Boolean heap. The relational product conjoins a Boolean heap with the abstract transition relation, projects the unprimed predicates, and renames primed to unprimed predicates in the resulting Boolean heap. Note that the abstract transition relation only depends on command $c$ and the context formula $\Gamma$. This allows us to cache abstract transition relations and avoid their recomputation in later fixpoint iterations if $\Gamma$ is unchanged.

**Splitting.** The Cartesian post operator maps each Boolean heap in a set of Boolean heaps to one Boolean heap. This means that in terms of precision the Cartesian post does not exploit the fact that the abstract domain is given by *sets* of Boolean heaps. In the following we describe an operation that splits a Boolean heap into a set of Boolean heaps. It is similar to the *focus* operation in TVLA [47]. Splitting maintains important invariants of Boolean heaps that result from best abstractions

of concrete states. We split Boolean heaps before applying the Cartesian post. This increases the precision of the analysis by carefully exploiting that the abstract domain is disjunctive complete.

Traditional shape analyses precisely keep track of objects which are pointed to by stack variables. This information is crucial for a precise analysis. In order to keep track of these objects we use abstraction predicates of the form $(x = v)$ where $x$ is some stack variable. Since these predicates denote singleton sets, i.e. each of them is true for exactly one object on the heap, we call them *singleton predicates*. Consequently, if a Boolean heap $H$ is the the result of applying the best abstraction with respect to $\gamma$ to some concrete state then for every singleton predicate $p$ it contains exactly one complete cube with a positive occurrence of $p$. Boolean heaps resulting from the Cartesian post might not have this property. This makes the analysis imprecise. Therefore we split each Boolean heap before application of the Cartesian post into a set of Boolean heaps, such that the above property is reestablished. Let $P$ be the subset of abstraction predicates denoting singletons then the *splitting operator* is defined as follows:

$$\begin{aligned}
\mathsf{Split}(\mathcal{H}) &= \mathsf{split}(P, \mathcal{H}) \\
\mathsf{split}(\emptyset, \mathcal{H}) &= \mathcal{H} \\
\mathsf{split}(\{p\} \cup P', \mathcal{H}) &= \textbf{let } C_p = [p \mapsto 1] \textbf{ and } C_{\neg p} = [p \mapsto 0] \textbf{ in} \\
&\quad \textstyle\bigcup_{H \in \mathcal{H}} \mathsf{split}(P', \{\, H \sqcap \{C_{\neg p}\} \sqcup \{C\} \mid C \in_c (H \sqcap \{C_p\}) \,\}).
\end{aligned}$$

The splitting operator takes a set of Boolean heaps $\mathcal{H}$ as arguments. For each singleton predicate $p$ and Boolean heap $H$ it splits $H$ into a set of Boolean heaps. Each of the resulting Boolean heaps corresponds to $H$, but contains only one of the complete cubes in $H$ that have a positive occurrence of $p$. The splitting operator is sound, i.e. satisfies: $\gamma(\mathsf{Split}(\mathcal{H})) = \gamma(\mathcal{H})$.

**Cleaning.** Splitting might introduce unsatisfiable Boolean heaps, because it is done propositionally without taking into account the semantics of predicates. Unsatisfiable Boolean heaps potentially lead to spurious counterexamples and hence should be eliminated. The same applies to cubes that are unsatisfiable with respect to other cubes within one Boolean heap. We use a *cleaning operator* [1] to eliminate unsatisfiable Boolean heaps and unsatisfiable cubes within satisfiable Boolean heaps. At the same time we strengthen the Boolean heaps with the guard of the commands before the actual computation of the Cartesian post. The cleaning operator is defined as follows:

$$\begin{aligned}
\mathsf{Clean}(F, \mathcal{H}) = \textbf{let } \mathcal{H}_1 = \{\, H \in \mathcal{H} \mid F \wedge \gamma(H) \not\models \mathsf{false} \,\} \textbf{ in} \\
\{\, \{\, C \in_c H \mid F \wedge \gamma(H) \wedge \gamma(C) \not\models \mathsf{false} \,\} \mid H \in \mathcal{H}_1 \,\}.
\end{aligned}$$

The operator $\mathsf{Clean}$ takes as arguments a formula $F$ (e.g. the guard of a command) and a set of Boolean heaps. It first removes all Boolean heaps that are unsatisfiable with respect to $F$. After that it removes from each remaining Boolean heap $H$ all complete cubes which are unsatisfiable with respect to $F$ and $H$. The cleaning

[1] The cleaning operator resembles the *coerce* operation in TVLA [47].

$$\mathsf{abstract}(F) = \textbf{let } H = \overline{\{\, C \mid C \models \neg F \,\}} \textbf{ in } \mathsf{Clean}(F, \mathsf{Split}(H))$$

$\textbf{proc } \mathsf{AbstractPost}(c,\ \mathsf{context} : \mathsf{AbsDom},\ \mathcal{H}_0 : \mathsf{AbsDom}) : \mathsf{AbsDom} =$
  $\textbf{let } \mathcal{H} = \mathsf{Clean}(\mathsf{guard}(c), \mathsf{Split}(\mathcal{H}_0))$
  $\textbf{let } \Gamma = \kappa(\mathsf{context} \sqcup \mathcal{H})$
  $\textbf{return } \mathsf{CSCartesianPost}(c, \Gamma, \mathcal{H})$

Fig. 7. Bohne's abstract post operator

operator is sound, i.e. strengthens $\mathcal{H}$ with respect to $F$:

$$F \wedge \gamma(\mathcal{H}) \models \gamma(\mathsf{Clean}(F, \mathcal{H})) \models \gamma(\mathcal{H}) \ .$$

Obviously the cleaning and splitting operators bear the danger of an exponential blowup. This can be avoided, e.g. by giving up precision and enforcing a polynomial bound by only considering cubes up to a fixed length. However, in practice this does not seem to be necessary, because Boolean heaps are relatively sparse and contain only few complete cubes.

**Abstract post operator.** Figure 7 defines the abstract post operator used in Bohne. It is defined as the composition of the splitting, cleaning, and the Cartesian post operator. The function $\kappa$ is a *context operator*. A context operator is a monotone mapping from sets of Boolean heaps to a context formula. It controls the trade-off between precision and efficiency of the abstract post operator. Our choice of $\kappa$ is described in the next section. Figure 7 also defines the abstraction function that is used to compute the initial set of Boolean heaps. For abstracting a formula $F$ the function abstract first computes a Boolean heap $H$ which is the complement of an under-approximation of $\neg F$. It then splits $H$ with respect to singleton predicates and strengthens the result by the original formula $F$. We compute the abstraction indirectly because it allows us to reuse all the functionality that we need for computing the abstract post operator. We also avoid computing the best abstraction function for the abstract domain, because the computational overhead is not justified in terms of the gained precision.

Assuming that $\kappa$ is in fact a context operator, soundness of AbstractPost follows from the soundness of all its component operators. Note that soundness is still guaranteed if the underlying validity checker is incomplete.

## 4 Context Instantiation

The context information used to strengthen the abstraction is given by the set of Boolean heaps that are already discovered at the respective program location. If we take into account all available context for the abstraction of a transition then we need to recompute the abstract transition relation in every iteration of the fixed point computation. Otherwise the analysis would be unsound. In order to avoid unnecessary recomputations we use the operator $\kappa$ to abstract the context by a context formula that less likely changes from one iteration to the next. For this purpose we introduce a domain-specific quantifier instantiation technique. We use this

$$\begin{aligned}
&\mathsf{Var} - \text{object-valued program variables}\\
&\mathsf{instantiate}(H : \text{Boolean heap}) : \text{formula} =\\
&\quad \textbf{let } \mathsf{cube}(x) = \bigsqcup (H \sqcap \{[(x = v) \mapsto 1]\}) \textbf{ in}\\
&\quad \bigwedge_{x \in \mathsf{Var}} \gamma(\mathsf{cube}(x))[v := x]\\
&\kappa(\mathcal{H}) = \textbf{let } H = \bigsqcup \mathcal{H} \textbf{ in } \mathsf{instantiate}(H)
\end{aligned}$$

Fig. 8. Context instantiation and the context operator $\kappa$

technique not only in connection with the context operator, but more generally to eliminate any universal quantifier in a decision procedure query that originates from the concretization of a Boolean heap. This eliminates the need for the underlying decision procedures to deal with quantifiers.

We observed that the most valuable part of the context is the information available over objects pointed to by program variables. This is due to the fact that transitions always change the heap with respect to these objects. We therefore instantiate Boolean heaps to objects pointed to by stack variables. Bohne automatically adds an abstraction predicate of the form $(x = v)$ for every object-valued program variable $x$. A syntactic backwards analysis of the procedure's assert statements and postcondition is used to determine which of these predicates are relevant at each program point.

Figure 8 defines the function instantiate. It uses the above mentioned predicates to instantiate a Boolean heap $H$ to a quantifier free formula (assuming predicates itself are quantifier free). For every program variable $x$ it computes the least upper bound of all cubes in $H$ which have a positive occurrence of predicate $(x = v)$. The resulting cube is concretized and the free variable $v$ is substituted by program variable $x$. The function $\kappa$ maps a set of Boolean heaps $\mathcal{H}$ to a formula by taking the join of $\mathcal{H}$ and instantiating the resulting Boolean heap. One can show that $\kappa$ is indeed a context operator, i.e. $\kappa$ is monotone and the resulting formula is a context formula for $\mathcal{H}$.

## 5 Semantic Caching

Abstracting context does not avoid that abstract transition relations have to be recomputed occasionally in later fixpoint iterations. Whenever we recompute abstract transition relations we would like to reuse the results from previous abstractions. We do this on the level of decision procedure calls by caching the queries and the results of the calls. Syntactic caching of decision procedure queries has been used before (e.g. [2] mentions its use in the SLAM system [3]). The problem with simple syntactic caching of formulas in shape analysis is that the context formulae are passed to the decision procedure as part of the queries, so a simple syntactic approach is ineffective. However, the context consists of all discovered abstract states at the current iteration. Therefore, the context changes monotonically from one iteration to the next. The monotonicity of the context operator $\kappa$ guarantees that

context formulae, too, increase monotonically with respect to the entailment order. We therefore cache formulas by keeping track of the partial order on the context. Since context formulae occur in the antecedents of the queries, this allows us to reuse negative results of entailment checks from previous fixpoint iterations. This method is effective because in practice the number of entailments which are invalid exceeds the number of valid ones.

Furthermore, formulas are cached up to alpha equivalence. Since the cache is self-contained, this enables caching results of decision procedure calls not only across different fixpoint iterations for one procedure, but even across the analysis of different procedures. This yields substantial improvements for procedures that exhibit some similarity, which opens up the possibility of using our analysis in an interactive context. For example, we verified a procedure inserting an element into a sorted list (see `SortedList.add` in Figure 9) and repeated the analysis without erasing the cache on a modified version of the same procedure where two commuting assignments were exchanged. About 90% of the results to decision procedure calls were found in the cache, causing that running time went down from 11s to 3s.

# 6 Propagation of Precondition Conjuncts

It often happens that parts of loop invariants literally come from the procedure's preconditions. A common situation where this occurs is that a procedure executes a loop to traverse a data structure performing only updates on stack variables and after termination of the loop the data structure is manipulated. In such a case the data structure invariants are trivially preserved while executing the loop. Using an expansive symbolic shape analysis to infer such invariants is inappropriate. We therefore developed a fast but effective analysis that propagates conjuncts from the precondition across the procedure's control-flow graph. This propagation precedes the symbolic shape analysis, such that the latter is able to assume the previously inferred invariants.

The propagation analysis works as follows: it first splits the procedure's precondition into a conjunction of formulas and assumes all conjuncts at all program locations. It then recursively removes a conjunct $F$ at program locations that have an incoming control flow edge from some location where either (1) $F$ has been previously removed or (2) where $F$ is not preserved under post of the associated command. After termination of the analysis (none of the rules for removal applies anymore) the remaining conjuncts are guaranteed to be invariants at the corresponding program points.

The preservation of conjuncts is checked by discharging a verification condition (via decision procedure calls). The use of decision procedures makes this analysis more general than the syntactic approach for computing frame conditions for loops used in ESC/Java-like desugaring of loops [15]. In particular, the propagation is still applicable in the presence of heap manipulations that preserve the invariants in each loop-free code fragment. Unlike the Houdini tool [13], precondition conjunct propagation does not attempt to invent new predicates.

| benchmark | used DP | # predicates total (manually supplied) | # validity checker calls total (cache hits) | running time total (DP) |
|---|---|---|---|---|
| DLL.addLast | MONA | 7 (0) | 118 (19%) | 2s (69%) |
| List.reverse | MONA | 7 (2) | 371 (22%) | 4s (72%) |
| SortedList.add | MONA, CVC lite | 16 (1) | 368 (40%) | 11s (65%) |
| Skiplist.add | MONA | 20 (0) | 787 (44%) | 26s (57%) |
| Tree.add | MONA | 13 (0) | 358 (31%) | 31s (92%) |
| ParentTree.add | MONA | 13 (0) | 362 (32%) | 33s (91%) |
| Linear.arrayInv | CVC lite | 7 (5) | 882 (52%) | 57s (97%) |

Fig. 9. Results of Experiments

# 7 Experiments

We applied Bohne to verify operations on various data structures. Our experiments cover data structures such as singly-linked lists, doubly-linked lists, two-level skip lists, trees, trees with parent pointers, sorted lists, and arrays. The verified properties include: (1) absence of run-time errors, such as null pointer dereferences and array bound violations; (2) complex data structure consistency properties, such as preservation of the tree structure, array invariants, as well as sortedness; and (3) procedure contracts, stating e.g. how the set of elements stored in a data structure is affected by the procedure.

Figure 9 shows the results for a collection of benchmarks running on a 2 GHz Pentium M with 1 GB memory. The Jahob system is implemented in Objective Caml and compiled to native code. Running times include inference of loop invariants. This time dominates the time for a final check (using verification-condition generator) that the resulting loop invariants are sufficient to prove the postcondition. The benchmarks can be found on the Jahob project web page [27]. The version of Bohne used to generate these results uses a simple analysis of the source code to determine most of the abstraction predicates automatically; the number of predicates in parentheses indicates the additional predicates that we needed to specify to make the symbolic shape analysis sufficiently precise. Note that we did not need to specify how these predicates change in response to program statements; this is computed automatically by Bohne. Note also that our examples are not stand-alone programs that build and then traverse their own data structures. Instead, our examples use assume-guarantee reasoning of Jahob to verify procedures with non-trivial preconditions, postconditions and representation invariants. As a result, these examples can be used in the context of larger programs that are verified by more scalable analysis, as demonstrated in the Hob project [33].

We also examined the impact of context-sensitive abstraction and context instantiation on the running time of the analysis. The results are shown in Table 10. As expected, running times for context-sensitive abstraction with instantiation disabled are significantly higher (2-8 times) than with instantiation enabled. Without context instantiation abstract transition relations have to be recomputed many times and caching of decision procedure calls is less effective. If context-sensitive abstraction is disabled completely the analysis not only becomes less precise (e.g. the analysis failed to verify the SortedList and SkipList examples without context) but also in many cases slower. Most likely the less precise analysis needs to explore a larger part of the abstract state space.

| benchmark | DLL.addLast | SortedList.add | Skiplist.add | Tree.add |
|---|---|---|---|---|
| | no context ($\Gamma = \mathsf{true}$) | | | |
| running time | 2s | 14s | 29s | 71s |
| DP calls (cache hits) | 118 (20%) | 457 (32%) | 1110 (51%) | 1024 (51%) |
| | context-sensitive without instantiation ($\kappa = \mathsf{id}$) | | | |
| running time | 4s | 24s | 72s | 473s |
| DP calls (cache hits) | 178 (23%) | 445 (22%) | 1031 (38%) | 742 (13%) |
| | context-sensitive with instantiation | | | |
| running time | 2s | 11s | 26s | 32s |
| DP calls (cache hits) | 118 (19%) | 368 (40%) | 787 (44%) | 358 (31%) |

Fig. 10. Effect of context-sensitive abstraction and context instantiation

Note that our implementation of the algorithm is not highly tuned in terms of aspects orthogonal to Bohne's algorithm, such as type inference for internally manipulated Isabelle formulas. We expect that the running times would be notably improved using more efficient implementation of Hindley-Milner type reconstruction. In previous benchmarks without type reconstruction in average 97% of the time was spent in the decision procedures. The most promising directions for improving the analysis performance are therefore 1) deploying more efficient decision procedures, and 2) further reducing the number of decision procedure calls.

In addition to the presented examples, we have used the verification condition generator to verify examples such as array-based implementations of containers and implementations of association lists. Bohne can also infer loop invariants in such examples.

# 8 Conclusions

We described Bohne, a data structure verification algorithm based on symbolic shape analysis that infers invariants about sets given by predicates on objects. We showed how to fruitfully combine this abstraction with a collection of decision procedures that operate on independent subgoals of the same proof obligation. We deployed a range of techniques that improve the running time of the analysis and the level of automation compared to direct application of the algorithm. These techniques include context-dependent abstraction, semantic caching of formulas, propagation of conjuncts, and domain-specific quantifier instantiation. Our experience with the Bohne analysis in the context of the Hob and Jahob data structure verification systems suggests that it is effective for verifying a wide range of data structures with user-defined procedure contracts. The verified properties go beyond traditional shape properties such as treeness and include the characterization of data structure operations in terms of changes to their content.

# References

[1] Balaban, I., A. Pnueli and L. Zuck, *Shape analysis by predicate abstraction*, in: *VMCAI'05*, 2005.

[2] Ball, T., B. Cook, S. K. Lahiri and L. Zhang, *Zapato: Automatic theorem proving for predicate abstraction refinement*, in: *Tool Paper, CAV*, 2004.

[3] Ball, T., R. Majumdar, T. Millstein and S. K. Rajamani, *Automatic predicate abstraction of C programs*, in: *Proc. ACM PLDI*, 2001.

[4] Barrett, C. and S. Berezin, *CVC Lite: A new implementation of the cooperating validity checker*, in: *Proc.* $16^{th}$ *Int. Conf. on Computer Aided Verification (CAV '04)*, Lecture Notes in Computer Science **3114**, 2004, pp. 515–518.

[5] Bertot, Y. and P. Castéran, "Interactive Theorem Proving and Program Development–Coq'Art: The Calculus of Inductive Constructions," Springer, 2004.

[6] Beyer, D., T. A. Henzinger and G. Théoduloz, *Lazy shape analysis*, in: T. Ball and R. Jones, editors, *Proceedings of the 18th International Conference on Computer Aided Verification (CAV 2006, Seattle, WA, August 16-20)*, LNCS 4144 (2006), pp. 532–546.

[7] Bingham, J. and Z. Rakamarić, *A logic and decision procedure for predicate abstraction of heap-manipulating programs*, Technical Report TR-2005-19, UBC Department of Computer Science (2005).

[8] Bouillaguet, C., V. Kuncak, T. Wies, K. Zee and M. Rinard, *On using first-order theorem provers in a data structure verification system*, Technical Report MIT-CSAIL-TR-2006-072, MIT (2006), http://hdl.handle.net/1721.1/34874.

[9] Bryant, R. E., *Graph-based algorithms for boolean function manipulation*, IEEE Transactions on Computers **C-35** (1986), pp. 677–691.

[10] Calvanese, D., G. De Giacomo and M. Lenzerini, *Reasoning in expressive description logics with fixpoints based on automata on infinite trees*, in: *Proc. of the 16th Int. Joint Conf. on Artificial Intelligence (IJCAI'99)*, 1999, pp. 84–89.

[11] Cousot, P. and R. Cousot, *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints*, in: *Proc. 4th POPL*, 1977.

[12] Distefano, D., P. O'Hearn and H. Yang, *A local shape analysis based on separation logic*, in: *TACAS'06*, 2006.

[13] Flanagan, C. and K. R. M. Leino, *Houdini, an annotation assistant for esc/java*, in: *FME '01: Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity* (2001), pp. 500–517.

[14] Flanagan, C. and S. Qadeer, *Predicate abstraction for software verification*, in: *Proc. 29th ACM POPL*, 2002.

[15] Flanagan, C. and J. B. Saxe, *Avoiding exponential explosion: Generating compact verification conditions*, in: *Proc. 28th ACM POPL*, 2001.

[16] Fradet, P. and D. L. Métayer, *Shape types*, in: *Proc. 24th ACM POPL*, 1997.

[17] Georgieva, L. and P. Maier, *Description logics for shape analysis*, in: *Proc. 3rd SEFM*, 2005, pp. 321–330.

[18] Ghiya, R. and L. Hendren, *Is it a tree, a DAG, or a cyclic graph?*, in: *Proc. 23rd ACM POPL*, 1996.

[19] Ghiya, R. and L. J. Hendren, *Connection analysis: A practical interprocedural heap analysis for C*, in: *Proc. 8th Workshop on Languages and Compilers for Parallel Computing*, 1995.

[20] Gotsman, A., J. Berdine and B. Cook, *Interprocedural shape analysis with separated heap abstractions.*, in: *SAS*, 2006, pp. 240–260.
URL http://dx.doi.org/10.1007/11823230_16

[21] Grädel, E., *Decision procedures for guarded logics*, in: *Automated Deduction - CADE16. Proceedings of 16th International Conference on Automated Deduction, Trento, 1999*, LNCS **1632** (1999).
URL http://www-mgi.informatik.rwth-aachen.de/Publications/pub/graedel/Gr-cade99.ps

[22] Graf, S. and H. Saidi, *Construction of abstract state graphs with PVS*, in: *Proc. 9th CAV*, 1997, pp. 72–83.

[23] Henzinger, T. A., R. Jhala, R. Majumdar and G. Sutre, *Lazy abstraction*, in: *POPL*, 2002.

[24] Jones, N. D. and S. S. Muchnick, *Chapter 4: Flow analysis and optimization of LISP-like structures*, in: S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, Prentice Hall, 1981 .

[25] Klarlund, N., A. Møller and M. I. Schwartzbach, *MONA implementation secrets*, in: *Proc. 5th International Conference on Implementation and Application of Automata* (2000).

[26] Klarlund, N. and M. I. Schwartzbach, *Graph types*, in: *Proc. 20th ACM POPL*, Charleston, SC, 1993.

[27] Kuncak, V., *The Jahob project web page, http://www.mit.edu/∼vkuncak/projects/jahob/* (2006).

[28] Kuncak, V., "Modular Data Structure Verification," Ph.D. thesis, EECS Department, Massachusetts Institute of Technology (2007).

[29] Kuncak, V., P. Lam and M. Rinard, *Role analysis*, in: *Annual ACM Symp. on Principles of Programming Languages (POPL)*, 2002.

[30] Kuncak, V., H. H. Nguyen and M. Rinard, *Deciding Boolean Algebra with Presburger Arithmetic*, J. of Automated Reasoning (2006),
http://dx.doi.org/10.1007/s10817-006-9042-1.

[31] Lahiri, S. K. and R. E. Bryant, *Indexed predicate discovery for unbounded system verification*, in: *CAV'04*, 2004.

[32] Lahiri, S. K. and S. Qadeer, *Verifying properties of well-founded linked lists*, in: *POPL'06*, 2006.

[33] Lam, P., V. Kuncak and M. Rinard, *Hob: A tool for verifying data structure consistency*, in: *14th International Conference on Compiler Construction (tool demo)*, 2005.

[34] Lam, P., V. Kuncak, K. Zee and M. Rinard, *The Hob project web page*, http://hob.csail.mit.edu (2004).

[35] Lee, O., H. Yang and K. Yi, *Automatic verification of pointer programs using grammar-based shape analysis*, in: *ESOP*, 2005.

[36] Lev-Ami, T., "TVLA: A Framework for Kleene Based Logic Static Analyses," Master's thesis, Tel-Aviv University, Israel (2000).

[37] Lev-Ami, T., N. Immerman, T. Reps, M. Sagiv, S. Srivastava and G. Yorsh, *Simulating reachability using first-order logic with applications to verification of linked data structures*, in: *CADE-20*, 2005.

[38] Lev-Ami, T., T. Reps, M. Sagiv and R. Wilhelm, *Putting static analysis to work for verification: A case study*, in: *Int. Symp. Software Testing and Analysis*, 2000.

[39] Manevich, R., E. Yahav, G. Ramalingam and M. Sagiv, *Predicate abstraction and canonical abstraction for singly-linked lists*, in: R. Cousot, editor, *Proceedings of the 6th Int. Conf. on Verification, Model Checking and Abstract Interpretation, VMCAI 2005*, LNCS **3148** (2005), pp. 181–198.

[40] McPeak, S. and G. C. Necula, *Data structure specifications via local equality axioms*, in: *CAV*, 2005, pp. 476–490.

[41] Møller, A. and M. I. Schwartzbach, *The Pointer Assertion Logic Engine*, in: *Programming Language Design and Implementation*, 2001.

[42] Nipkow, T., L. C. Paulson and M. Wenzel, "Isabelle/HOL: A Proof Assistant for Higher-Order Logic," LNCS **2283**, Springer-Verlag, 2002.

[43] Podelski, A. and T. Wies, *Boolean heaps*, in: *Proc. Int. Static Analysis Symposium*, 2005.

[44] Ranise, S. and C. G. Zarba, *A decidable logic for pointer programs manipulating linked lists* (2005), http://cs.unm.edu/∼zarba/papers/pointers.ps.

[45] Reineke, J., "Shape Analysis of Sets," Master's thesis, Universität des Saarlandes, Germany (2005).

[46] Reps, T., M. Sagiv and A. Loginov, *Finite differencing of logical formulas for static analysis*, in: *Proc. 12th ESOP*, 2003.

[47] Sagiv, M., T. Reps and R. Wilhelm, *Parametric shape analysis via 3-valued logic*, ACM TOPLAS **24** (2002), pp. 217–298.

[48] Schulz, S., *E – A Brainiac Theorem Prover*, Journal of AI Communications **15** (2002), pp. 111–126.

[49] Sutcliffe, G. and C. B. Suttner, *The TPTP problem library: CNF release v1.2.1*, Journal of Automated Reasoning **21** (1998), pp. 177–203.

[50] Thatcher, J. W. and J. B. Wright, *Generalized finite automata theory with an application to a decision problem of second-order logic*, Mathematical Systems Theory **2** (1968), pp. 57–81.

[51] Voronkov, A., *The anatomy of Vampire (implementing bottom-up procedures with code trees)*, Journal of Automated Reasoning **15** (1995), pp. 237–265.

[52] Weidenbach, C., *Combining superposition, sorts and splitting*, , **II**, Elsevier Science, 2001 pp. 1965–2013.

[53] Wies, T., "Symbolic Shape Analysis," Master's thesis, Universität des Saarlandes, Saarbrücken, Germany (2004).

[54] Wies, T., V. Kuncak, P. Lam, A. Podelski and M. Rinard, *Field constraint analysis*, in: *Proc. Int. Conf. Verification, Model Checking, and Abstract Interpratation*, 2006.

[55] Yorsh, G., A. Rabinovich, M. Sagiv, A. Meyer and A. Bouajjani, *A logic of reachable patterns in linked data-structures*, in: *Proc. Foundations of Software Science and Computation Structures (FOSSACS 2006)*, 2006.

[56] Yorsh, G., T. Reps and M. Sagiv, *Symbolically computing most-precise abstract operations for shape analysis*, in: *10th TACAS*, 2004.

[57] Yorsh, G., T. Reps, M. Sagiv and R. Wilhelm, *Logical characterizations of heap abstractions*, TOCL **8** (2007).

[58] Yorsh, G., A. Skidanov, T. Reps and M. Sagiv, *Automatic assume/guarantee reasoning for heap-manupilating programs*, in: *1st AIOOL Workshop*, 2005.

# Inferring Local (Non-)Aliasing and Strings for Memory Safety [1]

Yannick Moy [2,3,5] Claude Marché [4,5]

*INRIA Futurs - ProVal*
*Parc Orsay Université - ZAC des Vignes*
*3, rue Jacques Monod - Bâtiment N*
*F-91893 ORSAY Cedex*

**Abstract**

We propose an original approach for checking memory safety of C pointer programs, by combining deductive verification and abstract interpretation techniques. The approach is modular and contextual, thanks to the use of Hoare-style annotations (pre- and postconditions), allowing us to verify each C function independently. Deductive verification is used to check these annotations in a sound way. Abstract interpretation techniques are used to automatically generate such annotations, in an *idiomatic* way: standard practice of C programming is identified and incorporated as heuristics.
Our first contribution is a set of techniques for identifying aliasing and strings, which we do in a local setting rather than through a global analysis as it is done usually. Our separation analysis in particular is a totally new treatment of non-aliasing. We present for the first time two abstract lattices to deal with local pointer aliasing and local pointer non-aliasing in an abstract interpretation framework. Our second contribution is the design of an abstract domain for implications, which makes it possible to build efficient contextual analyses. Our last contribution is an efficient back-and-forth propagation method to generate contextual annotations in a modular way, in the framework of abstract interpretation. We implemented our method in Caduceus, a tool for the verification of C programs, and successfully generated appropriate annotations for the C standard string library functions.

*Keywords:* C programming language, abstract interpretation, deductive verification, pointer programs, aliasing, buffer overflow, annotation inference

## 1 Introduction

Wrong memory usage is a major source of bugs in C programs, as exemplified by the well-known buffer overflow problem. Although many tools are designed to check the absence of threats on memory safety for C programs, none can ensure memory safety of real C programs used, e.g., in embedded devices. Moreover, existing tools often fail on quite simple programs, that an experienced C programmer would easily review for memory safety. This situation can be accounted for by the lack of support

[1] This research is partly supported by CIFRE contract 2005/973 with France Télécom company, and ANR RNTL CAT

[2] France Télécom, Lannion, France

[3] CNRS, Laboratoire de Recherche en Informatique, UMR8623, Orsay, F-91405

[4] INRIA Futurs, Orsay, F-91893

[5] Univ Paris-Sud, Orsay, F-91405

in these tools for widely spread C idioms, as well as the inability of the tools to understand comments embedded in the code, which express (some of) the invariants and preconditions that make programs correct.

To answer the second of these concerns, deductive verification is a good candidate. It aims at checking behavioral properties of programs, where behaviors are formally specified by logical annotations: function pre- and postconditions, code assertions, loop invariants, global invariants, etc. Verification tools based on deduction target mainly Java, annotated with Jml [19] or C#, annotated with the Spec# language [3]. On the bright side, deductive verification is very powerful: it is modular (each function can be verified independently), it is guaranteed to be sound and it allows describing complex behaviors of programs. On the dark side, annotations must be added manually by the programmer, which can be a very hard task. When analyzing standalone programs, abstract interpretation (abbrev. AI) provides a general and efficient solution to this problem. It consists in propagating an over-approximation of the sets of possible states forward through the control-flow graph. Sets of states are called abstract values and form abstract lattices, on which union and intersection are defined. Abstract domains are abstract lattices equipped with transfer functions which map programming statements to operations on abstract values. Abstract domain and concrete states are related through concretization and abstraction functions. Convergence is generally obtained through the use of widening operators. At each program point, AI generates an *invariant*, that is a valid formula at that point. This formula can then be seen as annotation in deductive verification [3] or as initial predicate in predicate abstraction [17]. Unfortunately, AI is not sufficient when doing modular verification: a forward analysis cannot generate function preconditions, which are essential for modularity.

The main contribution of this work is a method based on AI to automatically integrate the necessary annotations for modular deductive verification. Modularity is partly obtained by designing a contextual analyzer. Indeed, many C functions can be called in different valid contexts (think of the nullity of pointer arguments), with different behaviors and different assumptions. Merging the calling contexts of a function altogether is a crippling source of imprecision. This is why context-sensitivity is a key feature for verifying these programs. In whole-program analyses, the context of interest is taken to be the calling context, in a top-down analysis of the call-graph. Our approach focuses on the contexts needed by a function body to make programs correct, in a bottom-up approach. This is a key feature that allows modular verification. We successfully apply our method to programs with features that usually make the verification task difficult: pointer arithmetic, dynamic allocation, aliasing and strings. This is made possible only by incorporating into AI our knowledge of idiomatic C. In particular, we classically define strings as character pointers accessed up to a sentinel null character. We could easily accomodate other definitions of strings.

The remaining of this paper is organized as follows. Section 2 presents our framework for deductive verification and Section 3 our preliminary analyses. Section 4 describes our modular and contextual method for inferring annotations. Section 5 details the implementation and Section 6 reports experimental results. Section 7 discusses related work. We finally conclude and present future work in Section 8.

For conciseness reasons, we omit proofs here. More details can be found in [22].

## 2 Framework

Our background framework is given by the Caduceus tool [13] for deductive verification of C programs based on generation of weakest preconditions, using Hoare-style pre- and postconditions, loop invariants, etc. Like in Jml, such annotations are given as stylized comments.

To deal with pointer programs and aliasing, we consider the classical component-as-array modeling of heap [13,4]. This model replaces the physical view of heap as a large unstructured array by an unordered set of memory blocks. Fig. 1 gives the representation of such a block. For a given pointer $p$, a pointer addition $p+i$ is denoted $\mathit{shift}(p,i)$ on the model side. A logical function application $\mathit{arrlen}(p)$ denotes the number of cells between $p$ and the end of the block. Provided dereferencing $p$ is safe, which is guaranteed by adding the assertion $\mathit{arrlen}(p) > \mathit{0}$ as a guard, pointer dereferencing $\star p$ is modeled by $\mathit{select}(m,p)$, where $m$ is a variable called a heap component. The semantics of the model is defined through a first-order axiomatization detailed in [22].

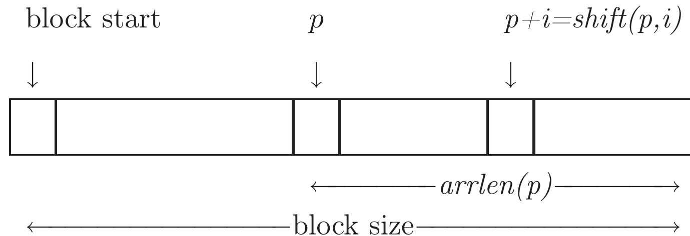



Fig. 1. Representation of a memory block

We focus on a subset of C that exhibits the kind of constructs that usually make automatic verification impossible: pointer arithmetic and aliasing. Current limitations concern the treatment of double indirection, casts, structures and memory deallocation. The examples of Fig. 2 will be used throughout this article to explain the non-obvious steps of the analysis. It consists in a simple yet idiomatic implementation of the `strcpy` and `memcpy` functions defined in C standard string library. We also show the kind of annotations that are classically added by hand in order to guarantee memory safety. As we will see later, these hand-written annotations are not accurate enough.

## 3 Aliasing and String Analyses

The first step of the method is a preprocess that removes most pointer arithmetic and some local aliasing, explicits some local non-aliasing and identifies strings. It is based on three original and simple independent analyses. We first classify each pointer value in the program into:

- a *base* pointer, a *cursor* pointer or a *complex* pointer. A base pointer at program label $L$ is either the initial value of a pointer parameter, or a pointer value returned by a call to some function on an execution path reaching $L$. A cursor pointer at program label $L$ is a pointer value that can be shown to be always aliased with

```
//@ requires arrlen(src) ≤ arrlen(dest)
char ⋆strcpy(char ⋆dest, char ⋆src) {
    char ⋆cur = dest;
    //@ invariant (0 ≤ cur − dest ≤ arrlen(old(src))) ∧ (cur − dest = src − old(src))
    while (⋆cur++ = ⋆src++) ;
    return dest;
}
//@ requires (n ≤ arrlen(src)) ∧ (n ≤ arrlen(dest))
char ⋆memcpy(char ⋆dest, char ⋆src, int n) {
    char ⋆cur = dest;
   //@ invariant (0 ≤ old(n) − n ≤ arrlen(old(src))) ∧ (cur − dest = old(n) − n = src − old(src))
    while (n−− > 0) ⋆cur++ = ⋆src++;
    return dest;
}
```

Fig. 2. Motivating examples

some other constant or variable offset expression from a same base pointer at $L$. We say the cursor pointer is *based* on the corresponding base pointer. E.g., if $p$ is a base pointer, $p+3$ and $p+f(q[i])$ are cursor pointers based on $p$. (C99 standard defines a similar notion of pointer *based* on an object when formally defining keyword *restrict* in §6.7.3.1., except it is a syntactic notion whereas ours is semantic.) A complex pointer at program label $L$ is a pointer that is neither a base pointer nor a cursor pointer.

- a *string* or a *(plain) pointer*. A string at program label $L$ is a pointer to an array of characters terminated by a sentinel, the null character. A (plain) pointer at program label $L$ is a pointer that is not a string.

### *3.1 Local aliasing*

It follows from the definition of cursor pointers that their occurrences in the program could be replaced by pointer expressions only mentioning base pointers. As such, identifying as many cursor pointers as possible is profitable to verification, since it decreases the number of pairs of pointers that could be aliased.

The analysis needed for this program transformation can be formalized as AI over some special pointer domain, presented in Fig. 3 with only two variables $v_1$ and $v_2$ and two integer constants $c_0$ and $c_1$, where elements point to their immediately greater element in the lattice ordering. In order to easily name base pointers, we introduce temporary variables to hold their value (as in static single assignment transformation). We divide cursor pointers into index ones and offset ones. Index pointers are aliased to some constant offset expression from a base pointer. Offset pointers are aliased to some variable expression from a base pointer. This leads to the introduction of integer offset variables to follow the value of this difference from the current base pointer (which may not be the same at all program points).

Local aliasing transformation uses the results of this analysis to remove pointer arithmetic, local pointer aliasing and pointer variables that it replaces with integer arithmetic, equality on integers and integer variables. Before we transform the program at program point $L$, where pointer variable $p$ is read or written, we need to take into consideration the abstract values associated to $p$ not only at $L$ but in all the program. We simply decided to lift all abstract values associated to $p$ to the topmost abstract value kind associated to $p$ in the program. Fig. 4 describes this

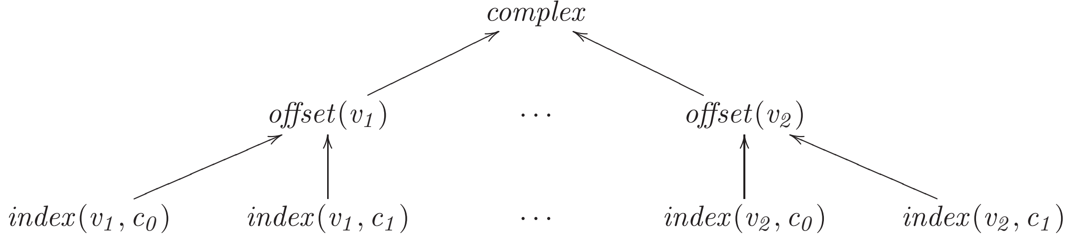



Fig. 3. Lattice for local aliasing

lifting in details.

| topmost kind | *index* | *offset* | *complex* |
|---|---|---|---|
| old abstract value | $index(bp, c)$ | $index(bp, _)$ or $offset(bp)$ | _ |
| new abstract value | $index(bp, c)$ | $offset(bp)$ | *complex* |

Fig. 4. Lifting abstract values

This transformation allows us to remove all the simple pointer arithmetic and some local aliasing, possibly all the local aliasing introduced by local variables in a function. The same analysis can also be used to rewrite pointer comparisons (resp. pointer differences) as integer ones when comparing (resp. subtracting) cursor pointers with the same base pointer (or a cursor pointer with its base pointer). This is a simple but crucial step, since it is common practice in C to use pointer arithmetic when iterating over an array for efficiency purposes. We do not detail it here, but this technique can also be applied on integer expressions to help discover relational invariants. Fig. 5 shows how this transformation affects the functions presented in Fig. 2.

```
char *strcpy(char *dest, char *src) {
    int src_self_offset = 0, cur_offset = 0;
    while (dest[cur_offset++] = src[src_self_offset++]) ;
    return dest;
}

char *memcpy(char *dest, char *src, int n) {
    int src_self_offset = 0, n_self_offset = 0, cur_offset = 0;
    while (n + n_self_offset-- > 0) dest[cur_offset++] = src[src_self_offset++];
    return dest;
}
```

Fig. 5. Examples (cont.): after local aliasing transformation

### *3.2 Local non-aliasing*

Section 3.1 did not deal with aliasing between different base pointers. This kind of aliasing cannot in general be inferred locally, but according to the following idiom, *non-aliasing* of base pointers can be.

**Idiom 1** *Given two different base pointers $p$ and $q$, a memory location written through some pointer expression based on $p$ must not be consequently read through some pointer expression based on $q$.*

This idiom expresses the local uniqueness of names to access modified memory locations. These names may not be unique in the original program, as far as they all refer to the same base pointer. It facilitates not only automated reasoning about the program but also human reasoning, which makes it important to follow in practice. The easiest way to guarantee that this idiom is respected when base pointer $p$ is read through after base pointer $q$ is written through is to ask for $p$ and $q$ to be *separated*. The need for annotations on pointer separation in a modular setting has been noted previously in [18]. Our contribution is to allow automatic inference of these annotations, using idiom 1. We introduce three related predicates to express this non-aliasing property:

- $separated(p, q)$ expresses that pointers $p$ and $q$ do not point to the same memory location. This is not the same as asking for $p \neq q$, since $p$ and $q$ may be equal as far as they do not point to some memory location. An important such case is when $p$ and $q$ are null.
- $full_separated(p, q)$ expresses that pointers $p$ and $q$ do not point to the same memory block. This is stronger than asking for simple separation. In our memory model, memory blocks allocated through different dynamic calls to some allocation function are not reachable one from the other. This makes the full separation of $p$ and $q$ equivalent to the separation of $p + i$ and $q + j$ for any integers $i$ and $j$:

$$full_separated(p, q) \equiv \forall\ int\ i, j.\ separated(p + i, q + j) \tag{1}$$

- $bound_separated(p, n, q, m)$ expresses that the memory chunks deliminated by $p$ included and $p + n$ excluded on one side, $q$ included and $q + m$ excluded on the other side, do not overloap. This is weaker than asking for full separation. Simple separation can be seen as a special case of bounded separation:

$$separated(p, q) \equiv bound_separated(p, 1, q, 1) \tag{2}$$

These relations between predicates can be formally stated as the lattice of Fig. 6, where p and q are pointer variables and $i$, $j$, $k$ and $l$ are integers (not necessarily integer constants). In our analysis, such an integer $i$ represents a constant or symbolic range $[0..i[$ of indices at which $p$ is read or written in the function.

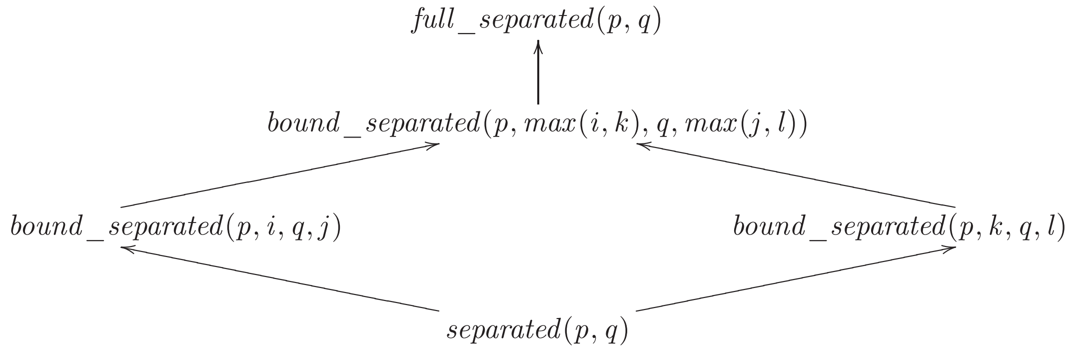



Fig. 6. Lattice of separation

Asking for any of these properties of base pointers can be formalized as a precondition or postcondition of functions, depending on the kind of base pointers involved.

This defines a transfer function on the call-graph, to be used in AI over the lattice of Fig. 6, generating preconditions and postconditions of functions along the way. On our motivating examples, this results in the same precondition for functions `strcpy` and `memcpy`, namely $\mathit{full_separated}(\mathit{dest}, \mathit{src})$. This amounts to asking for the non-overlapping of *src* reads and *dest* writes in the loop, which is the expected behavior.

### *3.3 Pointers and strings*

Plain pointers and strings are used in quite different ways in programs. Knowing which base pointers are strings and which are not allows us to infer useful information for each. Being a string is not a type information in C, since direct access to the string representation allows (re)moving the null sentinel character. This is more a typestate [28]. While a type is a predicate that characterizes an object throughout the program, a typestate is a predicate that characterizes an object at a precise location in the program. We approximate the string typestate using the following idiom.

**Idiom 2** *Character pointers whose value at some index is tested for nullity are strings.*

This gives us the seed information that we propagate backward to infer string pre- and postconditions on function parameters and function returns. We define a new predicate *string* for that purpose. Although the backward propagation is not sound, which is acceptable for an inference method, the forward propagation that follows it has to take into account the separation information to keep string information up-to-date. Overall, this back-and-forth method is a particular case of the method we describe in Section 4. We will detail its specificities after the general method has been presented.

In the memory model, we are entitled to define strings by the formula:

$$\begin{aligned} \mathit{string}(m, p) := \exists\ n.\ n < \mathit{arrlen}(p) \wedge \mathit{select}(m, \mathit{shift}(p, n)) = \mathit{0} \\ \wedge\ \forall\ i.\ \mathit{0} \leq i < n \rightarrow \mathit{select}(m, \mathit{shift}(p, i)) \neq \mathit{0} \end{aligned}$$

However, experiments with automatic provers show that this is not suitable for two reasons: first, provers do not easily handle the existential quantification; secondly, it is necessary to reason about unicity of length. Instead of a definition, we introduce in the model a new uninterpreted logical function *strlen*, so that $\mathit{strlen}(m, p)$ denotes the length of the string pointed-to by $p$. It defaults to $-1$ for a non-string pointer $p$, so that being a string can be expressed as

$$\mathit{string}(m, p) := 0 \leq \mathit{strlen}(m, p). \tag{3}$$

Given a plain pointer or a string $p$, we express the safety of a read or write access to $p$ as validity of a logical formula based on *arrlen* or *strlen*. We rely on the following idioms:

**Idiom 3** *A base pointer can only be read and written through at positive indices.*

**Idiom 4** *A string can only be read up to its sentinel null character.*

Idiom 3 is trivially true for fresh base pointers, that point by construction to the beginning of a memory block. After the local aliasing transformation, we expect it to be true of all base pointers in most programs. Idiom 4 is generally respected for all unbounded data structures terminated by a sentinel, in particular for strings. The safety of a read access $\star e$ for a string pointer expression $e$ is then formalized as $strlen(m, e) \geq 0$. Writing to a string is slightly more complex (see e.g., [2]). If $e$ is still expected to point to a string after the assignment, writing a non-null character is allowed only in the bounds of the string, which can be expressed as the formula $strlen(m, e) > 0$. Otherwise, writing is allowed in the bounds of the block pointed-to by $e$, as with a plain pointer, using formula $arrlen(e) > 0$.

These preliminary analyses modify our example code and add logical annotations to it, as shown in Fig. 7. Notice that heap components remain implicit in annotations.

```
//@ requires string(src) ∧ full_separated(dest, src)
char ⋆strcpy(char ⋆dest, char ⋆src) {
    int src_self_offset = 0, cur_offset = 0;
    //@ invariant string(src)
    while (dest[cur_offset++] = src[src_self_offset++]) ;
    return dest;
}

//@ requires full_separated(dest, src)
char ⋆memcpy(char ⋆dest, char ⋆src, int n) {
    int src_self_offset = 0, n_self_offset = 0, cur_offset = 0;
    while (n + n_self_offset−− > 0) dest[cur_offset++] = src[src_self_offset++];
    return dest;
}
```

Fig. 7. Examples (cont.): after aliasing and string analyses

## 4 Inferring Annotations by Abstract Interpretation

We design a very precise intra-procedural analysis, that is both flow-sensitive and path-sensitive in order to capture the possible complex data dependences that account for memory safety. To make it scalable despite its high local precision, we adopt a modular framework to communicate pre- and postconditions of functions throughout the call-graph. Finally, our analysis is rather *contextual* than context-sensitive: the context of interest for analyzing a function is not defined by the calling contexts (context-sensitive analysis) but by need, according to the function's body sensitivity to context (contextual analysis). Fig. 8 presents our method schematically.

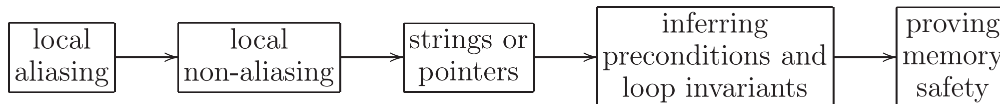



Fig. 8. Schematic view of our method

On the AI side, $arrlen(p)$ and $strlen(p)$ (without $m$ argument) are treated like meta-variables, that transfer functions must take into account. On the deductive verification side, *arrlen* and *strlen* are treated like uninterpreted functions, for which an appropriate axiomatization which does not use the existential quantification is given.

### *4.1 Implication Lattice*

Consider two abstract lattices $A$ and $B$, with appropriate union and intersection operations, that we denote respectively $\sqcup$ and $\sqcap$. We assume there is a Galois connection from the set $\Phi$ of first-order logic formulas without quantifiers to both $A$ and $B$ (as in [8]), where abstraction and concretization functions are denoted respectively *aval* and *pred*. An implication lattice $A \Rightarrow B$ of $A$ and $B$ is a lattice whose carrier $C_{A\Rightarrow B}$ is a subset of $A \times B$ such that any pair $(a, b)$ represents exactly the logical implication of the concretizations of $a$ and $b$. By definition, the following relation holds:

$$pred_{A\Rightarrow B}(a, b) \triangleq pred_A(a) \rightarrow pred_B(b). \tag{4}$$

To allow more efficient implementations, we do not require that the carrier is the full set $A \times B$. Instead, we extend the implication lattice over $A \times B$ by mapping any pair $(a, b)$ to a representative denoted $a \rightharpoonup b$ in the carrier such that:

$$pred_A(a) \rightarrow pred_B(b) \text{ logically implies } pred_{A\Rightarrow B}(a \rightharpoonup b). \tag{5}$$

This mapping is the identity on representatives, so that we identify $(a, b)$ and $a \rightharpoonup b$ on $C_{A\Rightarrow B}$. Take now $a_1 \rightharpoonup b_1$ and $a_2 \rightharpoonup b_2$ from $C_{A\Rightarrow B}$. We define a union and an intersection operations over $A \Rightarrow B$ as follows:

$$(a_1 \rightharpoonup b_1) \sqcup_{A\Rightarrow B} (a_2 \rightharpoonup b_2) \triangleq (a_1 \sqcap_A a_2) \rightharpoonup (b_1 \sqcup_B b_2), \tag{6}$$

$$(a_1 \rightharpoonup b_1) \sqcap_{A\Rightarrow B} (a_2 \rightharpoonup b_2) \triangleq (a_1 \sqcup_A a_2) \rightharpoonup (b_1 \sqcap_B b_2). \tag{7}$$

To ensure intersection correctly under-approximates conjunction, we ask that for all $a_1 \rightharpoonup b_1$ and $a_2 \rightharpoonup b_2$ in $C_{A\Rightarrow B}$:

$$(a_1 \sqcup_A a_2, b_1 \sqcap_B b_2) \in C_{A\Rightarrow B}. \tag{8}$$

Least and greatest elements are defined by $\bot_{A\Rightarrow B} \triangleq \top_A \rightharpoonup \bot_B$ and $\top_{A\Rightarrow B} \triangleq \bot_A \rightharpoonup \top_B$.

**Theorem 4.1** $A \Rightarrow B \triangleq (C_{A\Rightarrow B}, \bot_{A\Rightarrow B}, \top_{A\Rightarrow B}, \sqcup_{A\Rightarrow B}, \sqcap_{A\Rightarrow B})$ *forms a complete lattice.*

We assume there is a canonical implication form $\psi \rightarrow \phi$ for any formula $f$, such that $\psi = pred_A \circ aval_A(\psi)$. A trivial such formula is $True \rightarrow f$. Abstraction function $aval_{A\Rightarrow B}$ is defined by:

$$aval_{A\Rightarrow B}(\psi \rightarrow \phi) \triangleq aval_A(\psi) \rightharpoonup aval_B(\phi). \tag{9}$$

**Theorem 4.2** $(aval_{A\Rightarrow B}, pred_{A\Rightarrow B})$ *defines a Galois connection between* $\Phi$ *and* $A \Rightarrow B$.

If $B$ is stable by intersection (concretization of intersection is conjunction of concretizations), e.g., for the convex lattices we use most often in practice, we can also give an over-approximation $\Cap_{A\Rightarrow B}$ of the conjunction:

$$(a_1 \rightharpoonup b_1) \Cap_{A\Rightarrow B} (a_2 \rightharpoonup b_2) \triangleq (a_1 \sqcap_A a_2) \rightharpoonup (b_1 \sqcap_B b_2). \tag{10}$$

This operation allows strengthening invariants, which makes it generally more useful than the intersection operation. We note $\Cup_{A \Rightarrow B}$ the last useful operation that we can form:

$$(a_1 \rightharpoonup b_1) \Cup_{A \Rightarrow B} (a_2 \rightharpoonup b_2) \triangleq (a_1 \sqcup_A a_2) \rightharpoonup (b_1 \sqcup_B b_2). \tag{11}$$

This operation performs a union on both sides of the implication; we will see shortly why we need it. From now, for simplicity of exposure, we replace $A$ and $B$ in the implication lattice above by a unique abstract lattice $\mathcal{L}$ that we assume stable by intersection.

### *4.2 Inferring Loop Invariants and Function Preconditions*

In order to build the desired modular and contextual analysis, we use AI in a novel way described below. It consists in three propagation phases that are sketched in bold arrows on the diagrams in Fig. 9, where the last two phases are called in turn for each memory access.

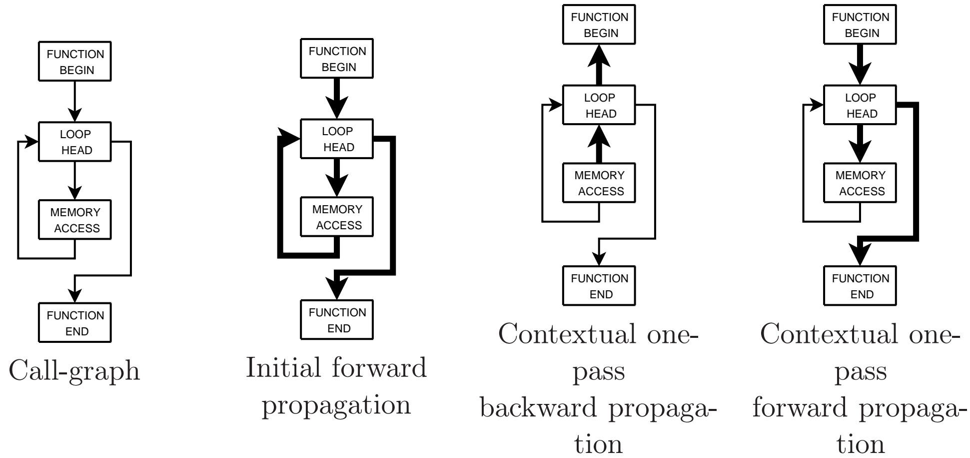



Fig. 9. Sketch of the method

#### *4.2.1 Initial Forward Propagation*

We use forward AI with widening as a first step to generate invariants at each program point. On our example, initial forward propagation with octagons as our background lattice $\mathcal{L}$ produces the annotated code shown in Fig. 10.

```
char *strcpy(char *dest, char *src) {
  int src_self_offset = 0, cur_offset = 0;
  //@ invariant (0 ≤ cur_offset = src_self_offset)
  while (dest[cur_offset++] = src[src_self_offset++]) ;
  return dest;
}

char *memcpy(char *dest, char *src, int n) {
  int src_self_offset = 0, n_self_offset = 0, cur_offset = 0;
  //@ invariant (0 ≤ cur_offset = src_self_offset = −n_self_offset)
  while (n + n_self_offset−− > 0) dest[cur_offset++] = src[src_self_offset++];
  return dest;
}
```

Fig. 10. Examples (cont.): after initial forward propagation

#### 4.2.2 Contextual One-Pass Backward Propagation

We consider each memory access in turn as a starting point for a one-pass backward propagation phase. Let $l$ be the label for this program point, $I$ the invariant abstract value computed by initial forward propagation at $l$ and $\phi$ the associated safety condition: if the formula $pred(I) \rightarrow \phi$ is valid, which we check in the abstract domain in our implementation, then the memory access is safe.

Otherwise, we use backward AI to propagate up in the code an over-approximation of the set of states from which it is possible to reach $l$ and an over-approximation of the set of states which result in validity of $\phi$ at $l$. If, at any point during this propagation, the former set of states is included in the latter, which was never approximated, then no execution from this point on can result in an unsafe memory access at $l$. Otherwise, we generate heuristically loop invariants and preconditions in order to prove the memory access safe. At $l$, the sets of states we want to propagate correspond respectively to $I$ and $\phi$. We see immediately that this propagation corresponds to operation $\Cup_{A \Rightarrow B}$ on the implication lattice $\mathcal{L} \Rightarrow \mathcal{L}$, starting from abstract value $aval_{\mathcal{L} \Rightarrow \mathcal{L}}(pred(I) \rightarrow \phi)$.

Two cases are possible once the abstract value representing an implication formula $\psi \rightarrow \phi$ reaches a loop head during backward propagation.

- (i) $\phi$ does not mention variables modified in the loop. We universally quantify those variables in $\psi$ and remove the quantifiers introduced. On octagons, it amounts to a *forget* operation, the usual backward transfer function for assignment.
- (ii) $\phi$ contains variables modified in the loop.

We define several elimination heuristics to deal with a variable $v$ in case ii. In our experiments with octagons, we noticed that applying Fourier-Motzkin whenever possible gives the best results. We use Fourier-Motzkin elimination in the cases where $v$ has a lower bound (or an upper bound) in both parts of the implication (as in [29]). These bounds are first-order logic terms, that can easily be made non-strict by incrementing or decrementing them by one, since our underlying type is integer. We rewrite the implication formula by only taking these bounds into account: $v \geq lhs_bound \rightarrow v \geq rhs_bound$. Validity of this equation where $v$ is universally quantified is equivalent to the following formula, obtained through Fourier-Motzkin elimination of $v$: $lhs_bound \geq rhs_bound$. We use it as new right-hand part of the implication. The formula obtained is added as new loop invariant. When our backward analysis reaches function beginning with abstract value $a \rightharpoonup b$, we add $pred_{A \Rightarrow B}(a \rightharpoonup b)$ as new function precondition. In our simple experiments, Fourier-Motzkin is usually sufficient to heuristically infer loop invariants and preconditions that guarantee memory safety.

#### 4.2.3 Contextual One-Pass Forward Propagation

Each one-pass backward propagation is followed by a one-pass forward propagation phase. First, it avoids performing backward propagation again from a memory access that could be proved safe using the newly generated logical annotations. Secondly, starting from the invariants computed by initial forward propagation, each forward propagation strengthens invariants, in particular loop invariants. Each invariant can

be written $pred_{\mathcal{L}}(I) \wedge \bigwedge_i pred_{\mathcal{L}\Rightarrow\mathcal{L}}(a_i \rightharpoonup b_i)$, where $a_i \rightharpoonup b_i$ are uniquely identified so that union and intersection are performed only on matching implication abstract values. This separates the main part $I$ of the invariant from its contextual parts.

The main part $I$ is propagated using forward AI without widening. To get a correct over-approximation on loops, we use the abstract value $J$ computed by the previous forward propagation phase at loop end to define a *one-pass refinement operator*, i.e. an operator which replaces the normal looping and widening process, so that iterating around loops is not needed anymore:

$$R_{\mathcal{L}}(I, J) \triangleq I \sqcup J. \tag{12}$$

In our context of use, new information forward propagated from a newly computed precondition or loop invariant is usually known to be true at loop end after initial forward propagation. This is because we use memory safety conditions as logical assertion during initial forward propagation. It makes our one-pass refinement operator very effective at discovering new loop invariants.

Contextual parts $a \rightharpoonup b$ are also propagated using forward AI without widening, in a way that minimizes the loss of information that occurs in unions. In our implementation with octagons, transfer functions compute an under-approximation of $a$ and an over-approximation of $b$, while adding as little inequalities in each. We devise a contextual variant of operator $R_{\mathcal{L}}$:

$$R_{\mathcal{L}\Rightarrow\mathcal{L}}(I, J, a \rightharpoonup b) \triangleq a \rightharpoonup aval_{\mathcal{L}}(pred_{\mathcal{L}}((I \sqcap a \sqcap b) \sqcup J) \setminus pred_{\mathcal{L}}(I)), \tag{13}$$

where $\setminus$ is a logical subtraction, such that $\phi \setminus \psi$ removes from the conjuncts of $\phi$ those conjuncts also in $\psi$.

**Theorem 4.3** *One-pass refinement operators $R_{\mathcal{L}}$ and $R_{\mathcal{L}\Rightarrow\mathcal{L}}$ correctly over-approximate the most precise loop invariant.*

In equation 13, the main part $I$ of the invariant is used to compute its contextual part $a \rightharpoonup b$. We do the opposite too. At any point, if $pred_{\mathcal{L}}(I)$ logically implies $pred_{\mathcal{L}}(a)$, then $pred_{\mathcal{L}}(b)$ is added to the invariant. This is somewhat similar to the operations performed in a reduced product. The declared objective of this step is to add enough information to the main part so that the newly computed invariant $I'$ implies the safety of the memory access that started this back-and-forth pass, or equivalently that $pred(I') \rightarrow \phi$ is valid.

### *4.3 Flashback: the case of strings*

As seen in equation 3, being a string is equivalent to some linear inequation involving the length of the string. This inequation can be represented in the abstract domain we work with, using $strlen(p)$ as a meta-variable, as seen in Section 3.3. Therefore, we can use the back-and-forth propagation just described to infer loop invariants and preconditions that are likely to guarantee a pointer is a string at some point in the program. On our example, string inference using backward propagation produces the annotated code shown in Fig. 11 (`memcpy` is not modified).

```
//@ requires (0 ≤ strlen(src))
char ⋆strcpy(char ⋆dest, char ⋆src) {
    int src_self_offset = 0, cur_offset = 0;
    //@ invariant (0 ≤ cur_offset = src_self_offset) ∧ (0 ≤ strlen(src))
    while (dest[cur_offset++] = src[src_self_offset++]) ;
    return dest;
}
```

Fig. 11. Example (cont.): after string inference (backward only)

Before propagating forward these preconditions and loop invariants, we should look more closely at what it means to be a string in C. Using idiom 4, we can add information on $strlen(s)$ whenever testing the (non-)nullity of some string access $s[i]$. Indeed, the nullity of $s[i]$ can be understood as the equality $i = strlen(s)$ and the non-nullity of $s[i]$ as the inequality $i < strlen(s)$. This can be added as an assumption on each branch that originates in a (non-)nullity test of $s[i]$.

On our example, the initial forward propagation produced $cur_offset \leq strlen(src)$ at loop end, and the current contextual forward propagation also gives invariant $cur_offset \leq strlen(src)$ at loop entry. Using the improved one-pass widening operator that we saw for initial forward propagation produces a generalized loop invariant $cur_offset \leq strlen(src)$.

### 4.4 Proving Memory Safety

Applying our back-and-forth method to our example produces the annotated code shown in Fig. 12, which, modulo our code transformation, strengthens the hand-written annotations given in Section 2. Caduceus [13] then translates C code into verification conditions, taking into account the necessary assertions to guarantee memory safety. On our example, the generated verification conditions are all proved by Simplify [10] and Yices [12].

```
//@ requires (0 ≤ strlen(src) < arrlen(dest)) ∧ full_separated(dest, src)
char ⋆strcpy(char ⋆dest, char ⋆src) {
    int src_self_offset = 0, cur_offset = 0;
    //@ invariant (0 ≤ cur_offset = src_self_offset)
    //@             ∧ (cur_offset ≤ strlen(src) < arrlen(dest))
    while (dest[cur_offset++] = src[src_self_offset++]) ;
    return dest;
}
//@ requires (0 < n → (n ≤ arrlen(src) ∧ n ≤ arrlen(dest))) ∧ full_separated(dest, src)
char ⋆memcpy(char ⋆dest, char ⋆src, int n) {
    int src_self_offset = 0, n_self_offset = 0, cur_offset = 0;
    //@ invariant (0 ≤ cur_offset = src_self_offset = −n_self_offset)
    //@             ∧ (0 < n → (n ≤ arrlen(src) ∧ n ≤ arrlen(dest)))
    while (n + n_self_offset−− > 0) dest[cur_offset++] = src[src_self_offset++];
    return dest;
}
```

Fig. 12. Examples (cont.): after back-and-forth inference

## 5 Implementation

In our implementation, we use the domain of octagons [21]. This minimal relational domain is able to express all constraints of the form $\pm x \pm y \leq c$ where $x$ and $y$ are variables and $c$ is an integer constant. This seems to be the most interesting domain

for memory analysis, as argued in [17], although we might need a full relational domain when considering programs with casts and unions. Our implementation (see http://caduceus.lri.fr) is roughly 10,000 lines of OCaml for the plugin part inside Caduceus, and a few hundred lines of C to patch the available octagon library [21].

The implication domain is based on the octagon one. Instead of defining the left part of the implication to be an octagon and the right part another octagon, we pack both parts in one octagon. This is an important decision both for scalability and simplicity. The drawback of this decision is that we cannot represent formulas like $x > 0 \rightarrow x > 10$, because the same inequality is used on both sides of the formula. This choice respects relations 5 and 8. In practice, we patched the octagon library to tag inequalities from the right part. One of the most important operations on octagons, the closure computation, has to be restricted to the untagged inequalities. Closing an octagon derives the tighest possible bounds on each considered inequality of the form $\pm x \pm y \leq c$. Our restriction prevents merging inequalities from the left and right parts of an implication.

## 6 Experiments

The standard string library as defined by Ansi C presents a good mix of pointer and string manipulations, with many implicit preconditions only given in textual description, like the overlapping conditions. Since available implementations are heavily optimized, using bit-field manipulations or assembly code, we hand-coded a forward implementation of <string.h> header file that would be both simple and idiomatic. On this implementation, we successfully generated the necessary annotations and automatically proved memory safety for 18 functions out of a total of 22. The four remaining functions are `strcat` and `strncat`, which require inferring linear inequations involving three variables (which is not possible with the octagon domain only) and `strtok` and `strerr`, which require inferring global invariants. With additional hand-written annotations, we also verified automatically these four functions.

As an example of an apparently complex yet correct precondition generated by our method, here is the precondition we generate for the function `strncpy` (which uses a logical function *min* not presented here but implemented):

//@ **requires** $(1 \leq n \rightarrow 0 \leq \mathit{strlen}(\mathit{src}))$
//@ $\quad\wedge\ (1 \leq n \wedge 0 \leq \mathit{strlen}(\mathit{src})) \rightarrow \mathit{min}(n-1, \mathit{strlen}(\mathit{src})) < \mathit{arrlen}(\mathit{dest})$
//@ $\quad\wedge\ (2 \leq n \rightarrow n \leq \mathit{arrlen}(\mathit{dest}))$
//@ $\quad\wedge\ \mathit{full_separated}(\mathit{dest}, \mathit{src})$

A quick case analysis on the value of $n$ leads to the equivalent simpler formula, which looks more like a valid precondition for `strncpy` that a programmer would specify:

//@ **requires** $(n \leq 0 \vee (0 \leq \mathit{strlen}(\mathit{src}) \wedge n \leq \mathit{arrlen}(\mathit{dest}))) \wedge\ \mathit{full_separated}(\mathit{dest}, \mathit{src})$

## 7 Related Work

Our work owes much to the early work of Bourdoncle [5]. He too focused on array bound checking. His backward propagation from assertions (the ones he calls *invariant assertions*) merges the conditions to reach the program point where the assertion is performed and the conditions to make this assertion valid. These are the parts we separate in our implication abstract value, which allows us to generate loop invariants and preconditions that precisely explicit programs latent specifications.

Another work close to ours, in its objectives and description, is the modular checker for buffer overflows of Hackett et al. [15]. Their annotation language based on *properties*, which would be uninterpreted functions in our setting, allows them to modularly check each function separately. Our contextual inference allows us to treat functions like `strcpy` and more importantly to infer different function preconditions for different contexts, both things their method cannot do and that they describe as *unannotatable interfaces*.

Our work applies the per-path summary approach of debugging tools like PREfix [6] or ARCHER [31] in a formally justified way amenable to verification. Various verification tools like BOON [30], CSSV [11] and Overlook [2] handle allocation and strings by using function symbols much like *arrlen* and *strlen*. We believe our approach is strictly more powerful due to its unique annotation inference method and handling of aliasing. Our implication lattice is only a special case of the reduced power domain defined in [9] and generalized in [14], but our implementation is the first practical implementation we know of.

The necessity for some logical specification of pointer separation in C dates back to C99 standard [16], with the addition of the *restrict* keyword. Various authors have described annotation-based systems to help programmers specify pointer separation [1,18]. This has been pushed forward as a kind of logic for shared mutable data structures by Reynolds in separation logic [25]. Our local separation analysis was inspired from these works, with the emphasis on locality and automatic inference.

Recently, there have been attempts at using a combination of various proof techniques. In [17], AI is used in a first phase to compute invariants about the program that are used in a second phase to seed predicate abstraction. Interestingly, they also use the octagon library [21]. In [20], a real feedback loop is built between AI and deductive verification. Although very promising, this approach suffers from the high cost of calling deductive verification repeatedly and it cannot generate preconditions.

Lastly, our backward inference bears some resemblance with the footprint analysis of Calcagno et al. [7]. But their goal is quite different from ours, which results in different methods. They focus on shape properties of C programs, while we focus on much simpler properties that are (usually) sufficient to prove memory safety.

## 8 Conclusion and Future Work

We presented a new static method for checking memory safety of pointer-intensive C programs. Our analysis is incomplete: it expects programs to follow commonly respected C idioms and only returns success for correct programs that follow the idioms we selected. Our method relies on an inference algorithm based on heuris-

tics, that combines forward and backward abstract interpretation, to generate the necessary logical annotations, mostly function preconditions and loop invariants. This algorithm was specifically designed for modular and contextual verification. In particular, we crafted a special implication domain for abstract interpretation. We showed how to implement it efficiently using a relational domain, which we did for the octagon domain. This makes the implication domain a cheap disjunctive domain for contextual analysis. We presented two previously unknown lattices for local pointer aliasing and local pointer non-aliasing. We showed that special symbols could be used both as meta-variable generators in abstract interpretation and as uninterpreted functions in deductive verification. This allowed collaboration between abstract interpretation and deductive verification as well as deductive verification of existential properties.

In order to deal with real programs, our next steps are the treatment of casts on one side and structures on the other side. Unions combine the difficulties of casts and structures. As argued in Section 5, we might need a full relational domain when considering programs with casts and unions, instead of the weaker octagon domain we have been using so far. Before doing so, we need to gather idioms on the ways C programmers use invariants to prevent memory errors when using casts, structures and unions. A crucial point here is that we need to check the proposed idioms on large bases of programs. We plan to do this through code instrumentation and runtime checking on typestates [28], much as what CCured does on types [23]. A starting point might be some interesting such idioms that have already been studied in the context of program understanding or shape analysis, e.g., in [27,24,26]. To scale better, we plan to verify memory safety using abstract interpretation alone whenever possible, and resort to deductive verification only in the most complex cases. This is allowed by the correctness of our forward propagation steps. Chang and Leino [8] presented a method based on forward abstract interpretation to infer properties on heap, mostly for object-oriented programs. We should look at ways to integrate it with our method.

Altogether, we showed our method could be used to prove memory safety of C pointer programs (those of the standard string library) that no other available tool can handle. We are looking forward to applying it to more programs.

*Acknowledgements*

We would like to thank all the people that contributed to unobfuscate this paper: Mathieu Baudet, Dariusz Biernacki, Nicolaj Bjørner, Sylvain Conchon, Jean-François Couchot, Jean-Christophe Filliâtre, Francesco Logozzo, Matteo Slanina. We also thank Antoine Miné for his octagon domain implementation. Finally, we thank Pierre Crégut for his decisive impulse and advisory work.

## References

[1] Aiken, A., J. S. Foster, J. Kodumal and T. Terauchi, *Checking and inferring local non-aliasing*, in: *Proc. PLDI '03*, New York, NY, USA, 2003, pp. 129–140.

[2] Allamigeon, X., W. Godard and C. Hymans, *Static analysis of string manipulations in critical embedded c programs.*, in: *SAS*, 2006, pp. 35–51.

[3] Barnett, M., B.-Y. E. Chang, R. DeLine, B. Jacobs and K. R. M. Leino, *Boogie: A modular reusable verifier for object-oriented programs*, fMCO 2005.

[4] Bornat, R., *Proving pointer programs in Hoare logic*, in: *Mathematics of Program Construction*, 2000, pp. 102–126.

[5] Bourdoncle, F., *Assertion-based debugging of imperative programs by abstract interpretation*, in: *Proc. ESEC '93*, London, UK, 1993, pp. 501–516.

[6] Bush, W. R., J. D. Pincus and D. J. Sielaff, *A static analyzer for finding dynamic programming errors*, Software Practice and Experience **30** (2000), pp. 775–802.

[7] Calcagno, C., D. Distefano, P. O'Hearn and H. Yang, *Footprint analysis: A shape analysis that discovers preconditions* (2007), invited lecture at HAV'07.

[8] Chang, B.-Y. E. and K. R. M. Leino, *Abstract interpretation with alien expressions and heap structures.*, in: *VMCAI*, 2005, pp. 147–163.

[9] Cousot, P. and R. Cousot, *Systematic design of program analysis frameworks*, in: *Proc. POPL'79*, 1979, pp. 269–282.

[10] Detlefs, D., G. Nelson and J. B. Saxe, *Simplify: a theorem prover for program checking.*, J. ACM **52** (2005), pp. 365–473.

[11] Dor, N., M. Rodeh and M. Sagiv, *Cssv: towards a realistic tool for statically detecting all buffer overflows in c*, in: *Proc. PLDI '03*, New York, NY, USA, 2003, pp. 155–167.

[12] Dutertre, B. and L. de Moura, "The YICES SMT Solver," Computer Science Laboratory, SRI International (2006), http://yices.csl.sri.com.

[13] Filliâtre, J.-C. and C. Marché, *Multi-prover verification of C programs*, in: *Proc. ICFEM'04*, LNCS **3308**, 2004, pp. 15–29.

[14] Giacobazzi, R. and F. Ranzato, *The reduced relative power operation on abstract domains*, Theor. Comput. Sci. **216** (1999), pp. 159–211.

[15] Hackett, B., M. Das, D. Wang and Z. Yang, *Modular checking for buffer overflows in the large*, in: *Proc. ICSE '06*, New York, NY, USA, 2006, pp. 232–241.

[16] International Organization for Standardization (ISO), "The ANSI C standard (C99)," http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1124.pdf.

[17] Jain, H., F. Ivancic, A. Gupta, I. Shlyakhter and C. Wang, *Using statically computed invariants inside the predicate abstraction and refinement loop.*, in: *Proc. CAV'06*, LNCS **4144**, 2006, pp. 137–151.

[18] Koes, D., M. Budiu and G. Venkataramani, *Programmer specified pointer independence*, in: *Proc. MSP '04*, New York, NY, USA, 2004, pp. 51–59.

[19] Leavens, G. T., K. R. M. Leino, E. Poll, C. Ruby and B. Jacobs, *JML: notations and tools supporting detailed design in Java*, in: *Proc. OOPSLA '00, Minnesota*, 2000, pp. 105–106.

[20] Leino, K. R. M. and F. Logozzo, *Loop invariants on demand.*, in: *APLAS*, 2005, pp. 119–134.

[21] Miné, A., *The octagon abstract domain*, Higher Order Symb. Comp. **19** (2006), pp. 31–100.

[22] Moy, Y. and C. Marché, *Checking C pointer programs for memory safety*, Technical report, LRI, Univ. Paris-Sud Orsay (2007), http://www.lri.fr/~moy.

[23] Necula, G. C., S. McPeak and W. Weimer, *Ccured: type-safe retrofitting of legacy code*, in: *Proc. POPL '02*, New York, NY, USA, 2002, pp. 128–139.

[24] Ranjit Jhala, R.-G. X., Rupak Majumdar, *State of the union: Type inference via craig interpolation*, in: *Proc. TACAS'07*, 2007.

[25] Reynolds, J., *Intuitionistic reasoning about shared mutable data structure*, in: *Millennial Perspectives in Computer Science* (2000).

[26] Shaunak Chatterjee, S. Q., Shuvendu K. Lahiri and Z. Rakamaric, *A reachability predicate for analyzing low-level software*, in: *Proc. TACAS'07*, 2007.

[27] Siff, M., S. Chandra, T. Ball, K. Kunchithapadam and T. Reps, *Coping with type casts in c*, in: *Proc. ESEC/FSE-7*, London, UK, 1999, pp. 180–198.

[28] Strom, R. E. and S. Yemini, *Typestate: A programming language concept for enhancing software reliability*, IEEE Trans. Softw. Eng. **12** (1986), pp. 157–171.

[29] Suzuki, N. and K. Ishihata, *Implementation of an array bound checker*, in: *Proc. POPL '77*, New York, NY, USA, 1977, pp. 132–143.

[30] Wagner, D., J. S. Foster, E. A. Brewer and A. Aiken, *A first step towards automated detection of buffer overrun vulnerabilities*, in: *NDSS Symposium*, San Diego, CA, 2000, pp. 3–17.

[31] Xie, Y., A. Chou and D. Engler, *Archer: using symbolic, path-sensitive analysis to detect memory access errors*, in: *Proc. ESEC/FSE-11*, New York, NY, USA, 2003, pp. 327–336.

# Reasoning about sequences of memory states (preliminary version)

Rémi Brochenin[2,3] Stéphane Demri[2,3] Etienne Lozes[2,3]

*LSV, ENS Cachan, CNRS, INRIA*
*61, av. Pdt. Wilson, 94235 Cachan Cedex, France*

**Abstract**

In order to verify programs with pointer variables, we introduce a temporal logic $\mathrm{LTL}^{\mathrm{mem}}$ whose underlying assertion language is the quantifier-free fragment of separation logic and the temporal logic on the top of it is the standard linear-time temporal logic LTL. We state the complexity of various model-checking and satisfiability problems for $\mathrm{LTL}^{\mathrm{mem}}$, considering various fragments of separation logic (including pointer arithmetic), various classes of models (with or without constant heap), and the influence of fixing the initial memory state. Our main decidability result is PSPACE-completeness of the satisfiability problems on the record fragment and on a classical fragment allowing pointer arithmetic. $\Sigma^0_1$-completeness or $\Sigma^1_1$-completeness results are established for various problems, and underline the tightness of our decidability results. This paper is a preliminary version of [4].

# 1 Introduction

**Verification of programs with pointers.**

Programs with pointer variables are worth to be analyzed and certified, since they easily contain programming errors that are sometimes difficult to detect. Such errors include the existence of memory leaks, memory violation, or undesirable aliasing. Prominent logics for analyzing such programs are Separation Logic [17], pointer assertion logic PAL [13], TVLA [14] and alias logic [3], to quote a few examples.

**Temporal Separation Logic: what for?**

Since [16], temporal logics are also used as languages for formal specification of programs. General and powerful automata-based techniques for verification have been developed, see e.g. [19]. On the other hand, Separation Logic is a static logic having a great success for program annotation [17], and more recently for symbolic computation [2]. Extending the scope of application of Separation Logic to standard

[1] Supported by a fellowship from CNRS/DGA.

[2] Work supported by the RNTL project "AVERILES".

[3] `{brocheni,demri,lozes}@lsv.ens-cachan.fr`

temporal logic-based verification technique has many potential interests, either for model-checking programs, or for defining restricted forms of recursive predicates. For instance, if we write $\mathsf{X}\mathtt{x}$ to denote the next value of $\mathtt{x}$ (also sometimes written $\mathtt{x}'$), the formula $(\mathtt{x} \hookrightarrow \mathsf{X}\mathtt{x})\mathsf{U}(\mathtt{x} \hookrightarrow \mathtt{null})$, understood on a model with constant heap, characterises the existence of a simple flat list, which is usually written $\mu L(\mathtt{x}).\ \mathtt{x} \hookrightarrow \mathtt{null} \vee \exists \mathtt{x}'.\mathtt{x} \hookrightarrow \mathtt{x}' \wedge L(\mathtt{x}')$.

Temporal logics also allow to work in the very convenient framework of "programs-as-formulae" and decision procedures for logical problems can be directly used for program verification, see a standard reduction in [18]. For instance, the previous formula can be seen as a program walking on a list, and more generally programs without updates can be expressed as formulae. Some programs with update that perform a simple pass on the heap, have an input-output relation that may be described by a formula. For instance, the formula $(\mathtt{x} \hookrightarrow_0 \mathsf{X}\mathtt{x} \wedge \mathsf{X}\mathtt{x} \hookrightarrow_1 \mathtt{x})\mathsf{U}\mathtt{x} \hookrightarrow_0 \mathtt{null}$ expresses broadly that the list in the initial heap $h_0$ is reversed in the final heap $h_1$ (see also Sect. 4.2).

As a side interest, up to our knowledge, few decision procedures for programs working with pointer arithmetic have been proposed up to now, whereas arithmetical constraints in temporal logics are known to lead to undecidability, see e.g. [8]. Actually, there is a growing interest in understanding the interplay of pointer arithmetic, temporal reasonning, and non aliasing properties.

**Our contribution.**

In this note, we introduce a linear-time temporal logic $\mathrm{LTL}^{\mathrm{mem}}$ to specify sequences of memory states with underlying assertion language based on quantifier-free Separation Logic [17]. Our logic addresses a very general notion of models, including the aspects of pointer arithmetic and recursive structures with records. We distinguish the satisfiability problems from the model-checking problems, as well as distinct subclasses of interesting programs, like for instance the programs without destructive update. We have shown the pspace-completeness of the satisfiability problems SAT(CL) and SAT(RF) where CL is the classical fragment without separation connectives and RF is the record fragment with no pointer arithmetic but with separation connectives. This result is very tight, as both propositional LTL and static Separation Logic are already pspace-complete [18,6]. We have obtained these results by reduction to the nonemptiness problem for Büchi automata on an alphabet of symbolic memory states obtained by an abstraction that we have shown sound and complete. This is a variant of the automata-based approach introduced in [19] for plain LTL and further developed with concrete domains of interpretation in [9]. This result is not a direct consequence of the decidability of the state logic used in this temporal logic. For instance, Presburger arithmetic is decidable, but LTL with Presburger constraints is not. As a matter of fact, the abstraction method we use based on [15] does not scale to the whole temporal logic, due to a subtle interplay between separation connectives and pointer arithmetic. Similar techniques can be found in [12]. Observe that the satisfiability problem for the whole state logic (SL) is decidable. Moreover, we have obtained new undecidability results for several problems, for instance for $\mathrm{SAT}^{ct}(\mathrm{LF})$ (satisfiability with constant heap on the list fragment).

**Related work.**

Previous temporal logics designed for pointer verification include Evolution Temporal Logic [20], based on the three-valued logic abstraction method that made the success of TVLA [14], and Navigation temporal logic [10], based on a tableau method for model-checking quite similar to our automaton-based reduction. In these works, the assertion language for states is quite rich, as it includes for instance list predicate, quantification over adresses, and a freshness predicate. The price of this expressiveness is that only incomplete abstractions are proposed, whereas we stick to exact methods. More importantly, our work addresses models with constant heaps and pointer arithmetic, which has not been done so far, and leads to a quite different perspective.

This paper is a preliminary version of [4] even though Figure 1 contains few new results, see also a discussion in Sect. 4.

## 2 Memory model and specification language

We introduce below a separation logic dealing with pointer arithmetic and record values, and a temporal logic $\mathrm{LTL}^{\mathrm{mem}}$. Model-checking programs with pointer variables over $\mathrm{LTL}^{\mathrm{mem}}$ specifications is our main problem of interest.

### *2.1 A separation logic with pointer arithmetic*

**Memory states.**

First, let us introduce our model of memory. It captures features of programs with pointer variables that use pointer arithmetic and records. We assume a countably infinite set `Var` of variables (as usual, for a fixed formula we need only a finite amount), and an infinite set `Val` of values containing the set $\mathbb{N}$ of naturals, thought as address indexes, and a special value *nil*. For simplicity, we assume that $\mathtt{Val} = \mathbb{N} \uplus \{nil\}$. In order to model field selectors, we also consider some infinite set `Lab` of labels. In the remainder, we will assume some fixed injection $(\mathtt{x}, i) \in \mathtt{Var} \times \mathbb{N} \mapsto \langle \mathtt{x}, i \rangle \in \mathtt{Var}$.

We use the notation $E \rightharpoonup_{fin} F$ for the set of partial functions from $E$ to $F$ of finite domain; and $E \rightharpoonup_{fin+} F$ for the set of partial functions from $E$ to $F$ of finite and nonempty domain. The sets $\mathcal{S}$ of stores and $\mathcal{H}$ of heaps are then defined as follows: $\mathcal{S} \stackrel{\text{def}}{\equiv} \mathtt{Var} \rightarrow \mathtt{Val}$ and $\mathcal{H} \stackrel{\text{def}}{\equiv} \mathbb{N} \rightharpoonup_{fin} (\mathtt{Lab} \rightharpoonup_{fin+} \mathtt{Val})$. We call *memory state* a couple $(s, h) \in \mathcal{S} \times \mathcal{H}$.

We will refer to the domain of a heap $h$ by $\mathtt{dom}(h) \subseteq \mathbb{N}$. Intuitively, in our memory model, each index is thought as an entry point on some record cell containing several fields. Cells are either not allocated, or allocated with some record stored in. In a memory state $(s, h)$, the memory cell at index $i$ is *allocated* if $i \in \mathtt{dom}(h)$; in this case the stored record is $h(i) = \{l_1 \mapsto v_1, .., l_n \mapsto v_n\}$.

Note that the size of the information hold in a memory cell is not fixed, nor bounded. Our models could be more concrete considering labels as offsets and relying on pointer arithmetic. But for our purpose, it will be convenient to consider

| Expressions | State Formulae | |
|---|---|---|
| $e ::= \mathtt{x} \mid \mathtt{null}$ | $\mathcal{A} ::= \pi$ | |
| Atomic formulae | $\mid \mathcal{A} * \mathcal{B} \mid \mathcal{A} \mathrel{-\!\!*} \mathcal{B} \mid \mathtt{emp}$ | (spatial fragment) |
| $\pi ::= e = e' \mid e + i \overset{l}{\hookrightarrow} e$ | $\mid \mathcal{A} \wedge \mathcal{B} \mid \mathcal{A} \rightarrow \mathcal{A} \mid \top \mid \bot$ | (classical fragment) |

Satisfaction

| | |
|---|---|
| $(s,h) \models_{\mathrm{SL}} e = e'$ | iff $[\![ e ]\!]_s = [\![ e' ]\!]_s$, with $[\![ \mathtt{x} ]\!]_s = s(\mathtt{x})$ and $[\![ \mathtt{null} ]\!]_s = nil$ |
| $(s,h) \models_{\mathrm{SL}} e + i \overset{l}{\hookrightarrow} e$ | iff $[\![ e ]\!]_s \in \mathbb{N}$ and $[\![ e ]\!] + i \in \mathtt{dom}(h)$ and $h(s(\mathtt{x}) + i)(l) = [\![ e ]\!]_s$ |
| $(s,h) \models_{\mathrm{SL}} \mathtt{emp}$ | iff $\mathtt{dom}(h) = \emptyset$ |
| $(s,h) \models_{\mathrm{SL}} \mathcal{A}_1 * \mathcal{A}_2$ | iff $\exists\, h_1, h_2$ s.t. $h = h_1 * h_2$, $(s,h_1) \models_{\mathrm{SL}} \mathcal{A}_1$ and $(s,h_2) \models_{\mathrm{SL}} \mathcal{A}_2$ |
| $(s,h) \models_{\mathrm{SL}} \mathcal{A}' \mathrel{-\!\!*} \mathcal{A}$ | iff for all $h'$, if $h \perp h'$ and $(s,h') \models_{\mathrm{SL}} \mathcal{A}'$ then $(s, h * h') \models_{\mathrm{SL}} \mathcal{A}$ |
| $(s,h) \models_{\mathrm{SL}} \mathcal{A}_1 \wedge \mathcal{A}_2$ | iff $(s,h) \models_{\mathrm{SL}} \mathcal{A}_1$ and $(s,h) \models_{\mathrm{SL}} \mathcal{A}_2$ |
| $(s,h) \models_{\mathrm{SL}} \mathcal{A}' \rightarrow \mathcal{A}$ | iff $(s,h) \models_{\mathrm{SL}} \mathcal{A}'$ implies $(s,h) \models_{\mathrm{SL}} \mathcal{A}$ |
| $(s,h) \models_{\mathrm{SL}} \bot$ | never and $(s,h) \models_{\mathrm{SL}} \top$ always |

Table 1
The syntax and semantics of SL with pointer arithmetic and records

pointer arithmetic and recursive structures independently.

**Separation Logic.**

We now introduce the separation logic (SL), see e.g. [17], on top of which we will define our temporal logic. The syntax of the logic is given in Table 1. As separation logic is about reasoning on disjoint heaps, and we need to define what we mean by "disjoint heaps" in our model. We choose to allow to reason at the granularity of record cells, so that a record cell cannot be decomposed in disjoint parts. Let $h_1$ and $h_2$ be two heaps; we say that $h_1$ and $h_2$ are disjoint, noted $h_1 \perp h_2$, if $\mathtt{dom}(h_1) \cap \mathtt{dom}(h_2) = \emptyset$. The operation $h_1 * h_2$ is defined for disjoint heap as the *disjoint union* of the two partial functions. Semantics of formulae is defined by the satisfaction relation $\models_{\mathrm{SL}}$ (see Table 1).

Formulae $\pi$ are *atomic formulae*, in which $\mathtt{x} + i \overset{l}{\hookrightarrow} e$ states that the value of the $l$ field of the record stored at the address pointed by $\mathtt{x}$ with offset $i$ is equal to the value of the expression $e$. The formula $e = e'$ states the equality between two expressions, and $\mathtt{emp}$ means that the current heap has no memory cell allocated (empty heap).

Formulae $\mathcal{A}$ of SL are called *state formulae*. A formula $\mathcal{A} * \mathcal{B}$ with the *separation conjunction* states that $\mathcal{A}$ holds on some portion of the memory heap and $\mathcal{B}$ holds on a disjoint portion. A formula $\mathcal{A} \mathrel{-\!\!*} \mathcal{B}$ states that the current heap, when extended with any disjoint heap verifying $\mathcal{A}$, will verify $\mathcal{B}$.

In the remainder, we focus on several specific fragments of this separation logic. We say that a formula is in the *record fragment* (RF) if all subformulae $\mathtt{x} + i \overset{l}{\hookrightarrow} e$ use $i = 0$ (we then write $\mathtt{x} \overset{l}{\hookrightarrow} e$). We say that a formula is in the *classical fragment* (CL) if it does not use the connectives $*, -\!*$. Finally, we say that a formula is in the *list fragment* (LF) if it is in the classical fragment and all subformulae $\mathtt{x} + i \overset{l}{\hookrightarrow} e$ use $i = 0$ and $l = next$ (we simply write $\mathtt{x} \hookrightarrow e$). Clearly, the classical and record fragments are incomparable, while the list fragment is included in both of them.

Let us illustrate the expressive power of SL on examples. The formula $\neg\mathtt{emp} * \neg\mathtt{emp}$ means that at least two memory cells are allocated. The formula $\mathtt{x} \overset{l}{\mapsto} e$, defined as $\neg(\neg\mathtt{emp} * \neg\mathtt{emp}) \wedge \mathtt{x} \overset{l}{\hookrightarrow} e$, is the local version of $\mathtt{x} \overset{l}{\hookrightarrow} e$: $s, h \models_{\mathrm{SL}} \mathtt{x} \overset{l}{\mapsto} e$ iff $\mathsf{dom}(h) = \{s(\mathtt{x})\}$ and $h(s(\mathtt{x}))(l) = [\![ e ]\!]_s$. The formula $(\mathtt{x} \overset{l}{\hookrightarrow} \mathtt{null}) -\!* \perp$ is satisfied at $(s_0, h_0)$ whenever there is no heap $h_1$ with $h_1 \perp h_0$ that allocates the variable $\mathtt{x}$ to *nil* on $l$ field, that is $\mathtt{x}$ is allocated in $h_0$.

$\mathcal{A}$ is valid iff for every memory state $(s, h)$, we have $(s, h) \models_{\mathrm{SL}} \mathcal{A}$ (written $\models_{\mathrm{SL}} \mathcal{A}$). Satisfiability is defined dually.

**Proposition 2.1** *The model-checking, satisfiability and validity problems for* SL *are* PSPACE*-complete.*

PSPACE-hardness results are consequences of [7, Sect. 5.2].

### 2.2 Temporal extension

**Memory states sequences**

Models of the logic $\mathrm{LTL}^{\mathrm{mem}}$ are $\omega$-sequences of memory states, that is elements in $(\mathcal{S} \times \mathcal{H})^\omega$ and they are understood as infinite computations of programs with pointer variables. In order to analyze computations from programs without destructive update, we shall also consider models with constant heap, that is elements in $\mathcal{S}^\omega \times \mathcal{H}$.

**The logic $\mathrm{LTL}^{\mathrm{mem}}$.**

Formulae of $\mathrm{LTL}^{\mathrm{mem}}$ are defined in Table 2. Atomic formulae of $\mathrm{LTL}^{\mathrm{mem}}$ are state formulae from SL except that variables can be prefixed by the symbol "$\mathsf{X}$". For instance, $\mathsf{X}\mathtt{x}$ is interpreted by the value of $\mathtt{x}$ at the next memory state. The temporal operators are the standard next-time operator $\mathsf{X}$ and until operator $\mathsf{U}$ present in LTL, see e.g. [18]. The satisfaction relation $\rho, t \models \Phi$ where $\rho$ is a model of $\mathrm{LTL}^{\mathrm{mem}}$, $t \in \mathbb{N}$ and $\Phi$ is a formula is also defined in Table 2. We use standard abbreviations such as $\Diamond\Phi$, $\Box\Phi$ ...

Given a fragment F of SL, we write SAT(F) to denote the satisfiability problem for F: given a temporal formula $\phi$ in $\mathrm{LTL}^{\mathrm{mem}}$ with state formulae built over F, is there a model $\rho$ such that $\rho, 0 \models \phi$? The variant problem in which we require that the model has a constant heap [resp. that the initial memory state is fixed, say $(s, h)$] is denoted by $\mathrm{SAT}^{ct}(\mathrm{F})$ [resp. $\mathrm{SAT}_{init}(\mathrm{F})$]. The problem $\mathrm{SAT}^{ct}_{init}(\mathrm{F})$ is defined analogously.

Enriched expressions $\eta ::= \ \mathtt{x} \mid \mathsf{X}\eta \mid \ \mathtt{null}$

Atomic formulae $\pi ::= \ \eta = \eta' \mid \ \eta + i \overset{l}{\hookrightarrow} \eta'$

State formulae $\mathcal{A} ::= \ \pi \mid \ \mathtt{emp} \mid \ \mathcal{A} * \mathcal{B} \mid \ \mathcal{A} {-\!\!*} \mathcal{B} \mid \ \mathcal{A} \wedge \mathcal{B} \mid \ \mathcal{A} \rightarrow \mathcal{B} \mid \ \bot$

Temporal formulae $\Phi ::= \ \mathcal{A} \mid \ \mathsf{X}\Phi \mid \ \Phi \mathsf{U} \Phi' \mid \ \Phi \wedge \Phi' \mid \ \neg\Phi$

Semantics

$\rho, t \models \mathsf{X}\Phi$ iff $\rho, t+1 \models \Phi$.

$\rho, t \models \Phi \mathsf{U} \Phi'$ iff there is $t_1 \geq t$ s.t. $\rho, t_1 \models \Phi'$ and $\rho, t' \models \Phi$ for all $t' \in \{t, .., t_1 - 1\}$.

$\rho, t \models \Phi \wedge \Psi$ iff $\rho, t \models \Phi$ and $\rho, t \models \Psi$.

$\rho, t \models \neg\Phi$ iff $\rho, t \not\models \Phi$

$\rho, t \models \mathcal{A}$ iff $s'_t, h_t \models_{\mathrm{SL}} \mathcal{A}[\mathsf{X}^k \mathtt{x} \leftarrow (\mathtt{x}, k)]$ where $\rho = (s_t, h_t)_{t \geq 0}$ and

$s'_t$ is defined by $s'_t(\langle \mathtt{x}, k \rangle) = s_{t+k}(\mathtt{x})$.

Table 2
The syntax and semantics of $\mathrm{LTL}^{\mathrm{mem}}$

### *2.3 Programs with pointer variables*

In this section, we define the model-checking problems for programs with pointer variables over $\mathrm{LTL}^{\mathrm{mem}}$ specifications. The set $\mathtt{I}$ of *instructions* used in the programs is defined by the grammar below:

$$\begin{aligned} \mathtt{instr} ::= \ & \mathtt{x} := \mathtt{y} \mid \mathtt{skip} \\ & \mid \mathtt{x} := \mathtt{y} \rightarrow l \mid \mathtt{x} \rightarrow l := \mathtt{y} \mid \mathtt{x} := \mathtt{cons}(l_1 : x_1, .., l_k : x_k) \mid \mathtt{free}\ \mathtt{x} \\ & \mid \mathtt{x} := \mathtt{y}[i] \mid \mathtt{x}[i] := \mathtt{y} \mid \mathtt{x} = \mathtt{malloc}(i) \mid \mathtt{free}\ \mathtt{x}, i \end{aligned}$$

The denotational semantics of an instruction $\mathtt{instr}$ is defined as a partial function $[\![ \ \mathtt{instr} \ ]\!] : \mathcal{S} \times \mathcal{H} \rightarrow \mathcal{S} \times \mathcal{H}$, undefined when the instruction would cause a memory violation. We list in Table 3 the formal denotational semantics of our instruction set. Boolean combinations of equalities between expressions are called guards and their set is denoted by $G$. A program is defined as a triple $(Q, \delta, q_I)$ such that $Q$ is a finite set of control states, $q_I$ is the initial state and $\delta$ is the transition relation, a subset of $Q \times G \times \mathtt{I} \times Q$. We use $q \xrightarrow{g, \mathtt{instr}} q'$ to denote a transition. We say that a program is *without destructive update* if transitions are labeled only with instructions of the form $\mathtt{x} := \mathtt{y}$, $\mathtt{x} := \mathtt{y} \rightarrow l$, and $\mathtt{x} := \mathtt{y}[i]$.

A program is a finite object whose interpretation can be viewed as an infinite-state system. More precisely, given a program $\mathtt{p} = (Q, \delta, q_I)$, the transition system $\mathcal{S}_{\mathtt{p}} = (S, \rightarrow)$ is defined as follows:

- $S = Q \times (\mathcal{S} \times \mathcal{H})$ (set of configurations),

| | |
|---|---|
| $[\![\ \mathtt{x} := \mathtt{y}\ ]\!]\ (s,h)$ | $\stackrel{\text{def}}{\equiv}\ (s[\mathtt{x} \mapsto s(\mathtt{y})], h)$. |
| $[\![\ \mathtt{x} := \mathtt{y} \rightarrow l\ ]\!]\ (s, h * \{i \mapsto \{l \mapsto v, \dots\}\})$ | $\stackrel{\text{def}}{\equiv}\ (s[\mathtt{x} \mapsto v], h * \{i \mapsto \{l \mapsto v, \dots\}\})$ with $s(\mathtt{y}) = i$ |
| $[\![\ \mathtt{x} \rightarrow l := \mathtt{y}\ ]\!]\ (s, h * \{i \mapsto \{l \mapsto v, \dots\}\})$ | $\stackrel{\text{def}}{\equiv}\ (s, h * \{i \mapsto \{l \mapsto s(\mathtt{y}), \dots\}\})$ with $s(\mathtt{x}) = i$ |
| $[\![\ \mathtt{x} := \mathtt{cons}(l_1 : \mathtt{x}_1, .., l_k : \mathtt{x}_k)\ ]\!]\ \ (s,h)$ | $\stackrel{\text{def}}{\equiv}\ \big(s[\mathtt{x} \mapsto i], h * \{i \mapsto \{l_1 \mapsto s(\mathtt{x}_1), \dots, l_k \mapsto s(\mathtt{x}_k)\}\}\big)$ with $i \notin \mathtt{dom}(h)$ |
| $[\![\ \mathtt{free}\ \mathtt{x}, l\ ]\!]\ (s, h * \{i \mapsto \{l \mapsto v, \dots\}\})$ | $\stackrel{\text{def}}{\equiv}\ (s, h * \{i \mapsto \{\dots\}\})$ with $s(\mathtt{x}) = i$ |
| $[\![\ \mathtt{skip}\ ]\!]\ (s,h)$ | $\stackrel{\text{def}}{\equiv}\ (s,h)$ |
| $[\![\ \mathtt{x} := \mathtt{y}[i]\ ]\!]\ (s, h * \{i + i' \mapsto \{next \mapsto v\}\})$ | $\stackrel{\text{def}}{\equiv}\ (s[\mathtt{x} \mapsto v], h * \{i \mapsto \{next \mapsto v\}\}))$ with $s(\mathtt{y}) = i'$ |
| $[\![\ \mathtt{x}[i] := \mathtt{y}\ ]\!]\ (s, h * \{i' + i \mapsto \{next \mapsto v\}\})$ | $\stackrel{\text{def}}{\equiv}\ (s, h * \{i + i' \mapsto \{next \mapsto s(\mathtt{y})\}\})$ with $s(\mathtt{x}) = i'$ |
| $[\![\ \mathtt{x} := \mathtt{malloc}(i)\ ]\!](s,h)$ | $\stackrel{\text{def}}{\equiv}\ \big(s[\mathtt{x} \mapsto i'], h * \{i' \mapsto \{next \mapsto nil\}\} * \dots * \{i' + i \mapsto \{next \mapsto nil\}\big)$ with $i', .., i' + i \notin \mathtt{dom}(h)$ |
| $[\![\ \mathtt{free}\ \mathtt{x}, i\ ]\!]\ (s, h * \{i' + i \mapsto f\})$ | $\stackrel{\text{def}}{\equiv}\ (s,h)$ with $s(\mathtt{x}) = i'$ |

Table 3
Semantics for instructions

- $(q, (s,h)) \rightarrow (q', (s', h'))$ iff there is a transition $q \xrightarrow{g,\mathtt{instr}} q' \in \delta$ such that $(s,h) \models g$ and $(s', h') = [\![\ \mathtt{instr}\ ]\!](s,h)$.

Note that $\mathcal{S}_{\mathtt{p}}$ is not necessarily linear. A computation (or execution) of $\mathtt{p}$ is defined as an infinite path in $\mathcal{S}_{\mathtt{p}}$ starting with control state $q_I$. Computations of $\mathtt{p}$ can be viewed as $\mathrm{LTL}^{\mathrm{mem}}$ models, using propositional variables to encode the extra information about the control states.

Model-checking aims at checking properties expressible in $\mathrm{LTL}^{\mathrm{mem}}$ along computations of programs. To a logical fragment (SL, CL, RF, or LF), we associate a set of programs : all programs for SL and CL, programs with instructions having $i = 0$ for RF, and moreover with only the label *next* for LF. Given one of these fragments F of SL, we write MC(F) to denote the model-checking problem for F:

| | MC | $MC^{ct}$ | $MC^{ct}_{init}$ | $MC_{init}$ | SAT | $SAT^{ct}$ | $SAT^{ct}_{init}$ |
|---|---|---|---|---|---|---|---|
| LF | $\Sigma^1_1$-c. | $\Sigma^0_1$-c. | PSPACE-c. | $\Sigma^1_1$-c. | PSPACE-c. | $\Sigma^0_1$-c. | PSPACE-c. |
| CL and RF | $\Sigma^1_1$-c. | $\Sigma^0_1$-c. | PSPACE-c. | $\Sigma^1_1$-c. | PSPACE-c. | $\Sigma^0_1$-c. | PSPACE-c. |
| $SL\backslash\{-\!\!*\}$ | $\Sigma^1_1$-c. | $\Sigma^0_1$-c. | PSPACE-c | $\Sigma^1_1$-c. | $\Sigma^1_1$-c. | $\Sigma^0_1$-c. | PSPACE-c |
| SL | $\Sigma^1_1$-c. | $\Sigma^0_1$-c. | PSPACE-c | $\Sigma^1_1$-c. | $\Sigma^1_1$-c. | $\Sigma^1_1$-c. | $\Sigma^1_1$-c. |

Fig. 1. Complexity of reasoning about program with pointer variables

given a temporal formula $\phi$ in $LTL^{mem}$ with state formulae built over F and a program p of the associated fragment, is there an infinite computation $\rho$ of p such that $\rho, 0 \models \phi$ (which we write $\mathtt{p} \models \phi$)? The variant problem in which we require that the program is without destructive update [resp. that the initial memory state is fixed, say $(s,h)$] is denoted by $MC^{ct}(F)$ [resp. $MC_{init}(F)$]. The problem $MC^{ct}_{init}(F)$ is defined analogously. We may write $\mathtt{p}, (s,h) \models \phi$ to emphasize what is the initial memory state.

All the model-checking and satisfiability problems defined above can be placed in $\Sigma^1_1$ in the analytical hierarchy. Additionally, all the above problems can easily be shown PSPACE-hard since they all generalize LTL satisfiability and model-checking [18].

## 3 Complexity results

Figure 1 contains a summary of the complexity results about fragments of $LTL^{mem}$, more details can be found in [4]. However, let us present some ideas below. The decidability results for SAT(CL) and SAT(RF) are obtained thanks to a polynomial size abstraction of the memory states similar to symbolic heaps. A notable difference is that values of variables at successive states can be compared. This abstraction is used to define a finite alphabet on which is based a Büchi automaton recognizing a language of abstract runs. Unlike the standard construction for LTL, we need to recognize sequences of abstract memory states that admit a concrete sequence of memory states. Decidability is shown when the set of abstract sequences is $\omega$-regular.

By way of example, let us also consider the undecidability proof for SAT(SL). The recurrence problem for non-deterministic Minsky machines can be reduced to it. The main difficulty is to be able to encode incrementations and decrementations of a variable x. Observe that expressions of the form $\mathtt{x} = \mathtt{y} + 1$ do not belong to the logical language. The encoding will use in some essential way the interplay between the separation connectives and the temporal operators. We found two different ways to encode increment and decrement: using non-aliasing expressed by the separating conjunction, and using the precise pointing assertion $\mathtt{x} \overset{next}{\mapsto} \eta$ stating that the heap

contains only one cell, in conjunction with the $-\!\!*$ operator.

$$\phi^*_{\mathtt{x}++} = (\mathsf{X}\mathtt{x} \stackrel{next}{\hookrightarrow} \mathtt{null} \wedge \mathtt{x}+1 \stackrel{next}{\hookrightarrow} \mathtt{null}) \wedge \neg(\mathsf{X}\mathtt{x} \stackrel{next}{\hookrightarrow} \mathtt{null} * \mathtt{x}+1 \stackrel{next}{\hookrightarrow} \mathtt{null})$$
$$\phi^*_{\mathtt{x}--} = (\mathsf{X}\mathtt{x}+1 \stackrel{next}{\hookrightarrow} \mathtt{null} \wedge \mathtt{x} \stackrel{next}{\hookrightarrow} \mathtt{null}) \wedge \neg(\mathsf{X}\mathtt{x}+1 \stackrel{next}{\hookrightarrow} \mathtt{null} * \mathtt{x} \stackrel{next}{\hookrightarrow} \mathtt{null})$$
$$\phi^{-*}_{\mathtt{x}++} = \mathtt{emp} \quad \wedge \quad \big((\mathsf{X}\mathtt{x} \stackrel{next}{\mapsto} \mathtt{null}) -\!\!* \mathtt{x}+1 \stackrel{next}{\mapsto} \mathtt{null}\big)$$
$$\phi^{-*}_{\mathtt{x}--} = \mathtt{emp} \quad \wedge \quad \big((\mathtt{x} \stackrel{next}{\mapsto} \mathtt{null}) -\!\!* \mathsf{X}\mathtt{x}+1 \stackrel{next}{\mapsto} \mathtt{null}\big)$$

The formulae based on the separating conjunction correctly express incrementation and decrementation when the cells at index $\mathtt{x}, \mathtt{x}+1, \mathtt{x}-1$ are allocated, whereas formulae based on the operator $-\!\!*$ do not need the same assumption.

So, when the heap is constant, only the second way to encode increment applies. Moreover, in the absence of the operator $-\!\!*$, we can show that the problem $\mathrm{SAT}^{ct}_{init}(\mathrm{SL}\backslash\{-\!\!*\})$ is in PSPACE, which contrasts with the $\Sigma^1_1$-hardness of $\mathrm{SAT}^{ct}_{init}(\mathrm{SL})$. This PSPACE result is obtained by reduction into $\mathrm{SAT}^{ct}_{init}(\mathrm{RF})$. In some essential way, we take into account that the heap is constant and that only subheaps can be considered thanks to the absence of the operator $-\!\!*$. This decidability result also implies that $\mathrm{SAT}^{ct}(\mathrm{SL})$ is in $\Sigma^0_1$.

As far as model-checking problems are concerned, the complexity results do not depend on fragments. For the problems of the form $\mathrm{MC}^{ct}_{init}$, the program can be abstracted as a finite-state automaton. Using standard results for LTL and Proposition 2.1, we get the PSPACE upper bound. For the problems of the form $\mathrm{MC}^{ct}$, halting problem for Minsky machines can be encoded by guessing the maximal value of counters for reaching the halting state, whence the $\Sigma^0_1$-hardness results. Finally, the problems of the form MC can encode infinite runs of non-deterministic Minsky machines since the memory is not bounded, whence the $\Sigma^1_1$-hardness results.

## 4 Discussion

We provided above complexity results for reasoning tasks about $\mathrm{LTL}^{\mathrm{mem}}$. We are currently investigating issues about the expressive power of this logical formalism. Let us discuss few issues below.

First, the interest of model-checking programs with heap updates stems from early works on automata-based verification. Decision procedures are obtained at the cost of limitations: to define approximations as done in [20,10] or to restrict the programming language, see e.g. [1]. However, with this approach, compositionality principles are lost which is a pity since they made the success of separation logic, as frame rule and composition rule.

Second, assuming that the heap is constant is subject to promising development. Indeed, it is then possible to define spatial operators at the same syntactic level as temporal operators, and write formulae as e.g. $(\mathtt{x} \hookrightarrow \mathsf{X}\mathtt{x}\mathsf{U}\mathtt{x} \hookrightarrow \mathtt{null}) \; * \; y \mapsto \mathtt{null}$. This might be a way to model modularity in model-checking programs without destructive updates, but there are other points of interest we will try to advocate now.

### *4.1 Recursion with local parameters*

The constant heap semantics provides an original viewpoint for recursion with local parameters and local quantification. The problem of decision procedures in presence of recursive predicates has not yet completely satisfactory answers, as particular axiomatizations have been proposed for some standard recursive structures [5], or incomplete, though apparently good in practice, methods of inference.

In order to be a bit more precise, let us consider the fragment of recursive separation logic where all recursive formulae are of the form:

$$(1) \qquad \mu X(x_1,..,x_k).\ A(x_1,..,x_k) \vee \exists x'_1..x'_k.\ B(x_1,..,x_k,x'_1,..,x'_k) \wedge X(x'_1,..,x'_k)$$

This fragment is rich enough to express single lists, cyclic lists, and doubly-linked lists. However, we conjecture that it is not expressive enough for trees and DAGs.

We conjecture that deciding satisfiability in the fragment of recursive separation logic mentioned above reduces to $\mathrm{SAT}^{ct}(\mathrm{SL})$, and the model-checking problem reduces to $\mathrm{SAT}^{ct}_{init}$, considering that (1) can be rewritten as:

$$\big(B(x_1,..,x_k,\mathsf{X}x_1,..,\mathsf{X}x_k)\big)\ \mathsf{U}\ A(x_1,..,x_k).$$

In this perspective, our results could rise interesting decidability results for model-checking some of the recursive separation logic with local quantifiers. For satisfiability, we expect to define decidable fragments for $\mathrm{SAT}^{ct}(\mathrm{SL})$, for instance considering the techniques for checking temporal properties of flat programs without destructive updates introduced in [11]. Another interesting fragment of recursive separation logic is probably the one where recursion is guarded by the separation operator $*$, but we do not currently see how to treat it in the temporal logic perspective.

### *4.2 Programs as formulae*

Let us speculate some more. We may take advantage of expressing programs as formulae in order to reduce model-checking to satisfiability, a known approach from [18]. For programs without destructive update, we have the following result.

**Proposition 4.1** *Let* F *be a fragment of* SL *among* SL*,* CL*,* RF*, or* LF*. Then there is a logspace reduction from* $\mathrm{MC}^{ct}(\mathrm{F})$ *to* $\mathrm{SAT}^{ct}(\mathrm{F})$*.*

Intuitively, we translate instructions of the form $\mathtt{x} := \mathtt{y}$ into $\mathsf{X}\mathtt{x} = \mathtt{y}$, $\mathtt{x} := \mathtt{y} \rightarrow l$ into $\mathtt{y} \overset{l}{\hookrightarrow} \mathsf{X}\mathtt{x}$, and $\mathtt{x} := \mathtt{y}[i]$ into $\mathtt{y} + i \hookrightarrow \mathsf{X}\mathtt{x}$. Guards are translated accordingly. To translate the control of the program, we use special variables to encode the control state and define a formula that expresses the transition relation.

Moreover, we believe we can extend this result to programs with updates, but with a slightly different perspective. The constant heap semantics can be helpful to define the input-output relation of programs, even with destructive updates, provided some conditions on the way the program read and write over the memory are satisfied. To do so, we consider the extension of $\mathrm{LTL}^{\mathrm{mem}}$ with two predicates $\hookrightarrow_0$ and $\hookrightarrow_1$ instead of $\hookrightarrow$, and models are couples of state sequences with constant heap, that is tuples $\langle\ (s_i)_{i\geq 0}\ ,\ h_0\ ,\ h_1\rangle$. Let us define the input-output relation

$R_P$ of a program $P$ as : for all $(s_0, h_0), (s_1, h_1)$, $(s_0, h_0) R_P (s_1, h_1)$ if there is a run of $P$ that starts with $(s_0, h_0)$ and ends with $(s_1, h_1)$. Then we conjecture that for an interesting class of programs, this relation is definable in $\mathrm{LTL}^{\mathrm{mem}}$ extended with $\hookrightarrow_0$ and $\hookrightarrow_1$. Basically, the encoding of the control of the program will be the same as for programs without destructive updates, but the encoding of the instructions will be different. For instance, $\mathtt{x} \rightarrow l := \mathtt{y}$ would be encoded by $\mathtt{y} \overset{l}{\hookrightarrow}_1 \mathsf{X}\mathtt{x}$, whereas $\mathtt{x} := \mathtt{y} \rightarrow l$ would be encoded as $(\mathsf{X}\mathtt{x}) \overset{l}{\hookrightarrow}_0 \mathtt{y}$

## 5 Conclusion

We have introduced a logic that combines both aspects of temporal logic and separation logic, and permits to express constraints between values at different instants. We defined several decision problems, and studied their decidability and complexity. One of the most important complexity results states the decidability of the satisfiability problem without pointer arithmetic or with pointer arithmetic but without separation operators. In those cases, the problems are in PSPACE, which is really tight considering that both SL and LTL satisfiability problems are PSPACE-complete. This is not completely expected since LTL with simple Presburger constraints is undecidable even though both LTL and expressive fragments of Presburger arithmetic can be solved in PSPACE. Our result strongly relies on a faithful abstraction of memory states. As a future work, we plan to investigate decidable fragments of separation logic with recursion.

## References


[1] Bardin, S., A. Finkel, E. Lozes and A. Sangnier, *From pointer systems to counter systems using shape analysis*, 5th International Workshop on Automated Verification of Infinite-State Systems (AVIS'06) (2006).

[2] Berdine, J., C. Calcagno and P. W. O'Hearn, *Symbolic execution with separation logic*, APLAS'05 **3780** (2005), pp. 52–68.

[3] Bozga, M., R. Iosif and Y. Lakhnech, *On logics of aliasing*, in: *SAS'04*, LNCS **3148** (2004), pp. 344–360.

[4] Brochenin, R., S. Demri and E. Lozes, *Reasoning about sequences of memory states*, in: *LFCS'07*, LNCS (2007), to appear.

[5] Calcagno, C., J. Berdine and P. O'Hearn, *Smallfoot: Modular automatic assertion checking with separation logic* **4111** (2005), pp. 115–137.

[6] Calcagno, C., H. Yang and P. O'Hearn, *Computability and complexity results for a spatial assertion language*, in: *APLAS'01*, 2001, pp. 289–300.

[7] Calcagno, C., H. Yang and P. O'Hearn, *Computability and complexity results for a spatial assertion language for data structures*, in: *FST&TCS'01*, LNCS **2245** (2001), pp. 108–119.

[8] Comon, H. and V. Cortier, *Flatness is not a weakness*, CSL'00 **1862** (2000), pp. 262–276.

[9] Demri, S. and D. D'Souza, *An automata-theoretic approach to constraint LTL*, Information and Computation **205** (2007), pp. 380–415.

[10] Distefano, D., J.-P. Katoen and A. Rensink, *Who is pointing when to whom? on the automated verification of linked list structures*, in: *FSTTCS'04*, LNCS **3328** (2004), pp. 250–262.

[11] Finkel, A., E. Lozes and A. Sangnier, *Towards model-checking pointer systems without destructive update*, 2007, under submission.

[12] Galmiche, D. and D. Mery, *Characterizing provability in BI's pointer logic through resource graphs*, in: *LPAR'05*, LNCS **3835** (2005), pp. 459–473.

[13] Jensen, J., M. Jorgensen, N. Klarlund and M. Schwartzbach, *Automatic verification of pointer programs using monadic second-order logic*, in: *PLDI'97* (1997), pp. 226–236.

[14] Lev-Ami, T. and M. Sagiv, *TVLA: A system for implementing static analyses*, in: *SAS'00*, 2000, pp. 280–301.

[15] Lozes, E., *Separation logic preserves the expressive power of classical logic*, in: *2nd Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management (SPACE'04)*, 2004.

[16] Pnueli, A., *The temporal logic of programs*, in: *FOCS'77* (1977), pp. 46–57.

[17] Reynolds, J., *Separation logic: a logic for shared mutable data structures*, in: *LICS'02* (2002), pp. 55–74.

[18] Sistla, A. and E. Clarke, *The complexity of propositional linear temporal logic*, JACM **32** (1985), pp. 733–749.

[19] Vardi, M. and P. Wolper, *Reasoning about infinite computations*, Information and Computation **115** (1994), pp. 1–37.

[20] Yahav, E., T. Reps, M. Sagiv and R. Wilhelm, *Verifying temporal heap properties specified via evolution logic*, in: *ESOP'03*, LNCS **2618** (2003), pp. 204–22.

# Liveness of Heap Data for Functional Programs

Amey Karkare[1] Uday Khedker Amitabha Sanyal

{karkare,uday,as}@cse.iitb.ac.in
*Department of CSE, IIT Bombay*
*Mumbai, India*

**Abstract**

Functional programming languages use garbage collection for heap memory management. Ideally, garbage collectors should reclaim all objects that are *dead* at the time of garbage collection. An object is dead at an execution instant if it is not used in future. Garbage collectors collect only those dead objects that are not reachable from any program variable. This is because they are not able to distinguish between reachable objects that are dead and reachable objects that are live.
In this paper, we describe a static analysis to discover reachable dead objects in programs written in first-order, eager functional programming languages. The results of this technique can be used to make reachable dead objects unreachable, thereby allowing garbage collectors to reclaim more dead objects.

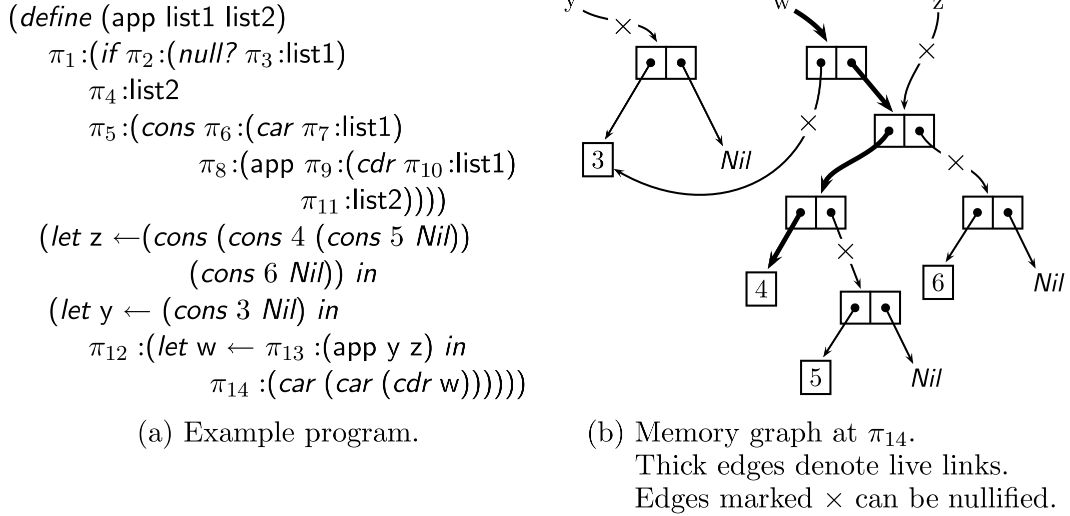
*Keywords:* Compilers, Liveness, Garbage Collection, Memory Management, Data Flow Analysis, Context Free Grammars

## 1 Introduction

Garbage collection is an attractive alternative to manual memory management because it frees the programmer from the responsibility of keeping track of object lifetimes. This makes programs easier to implement, understand and maintain. Ideally, garbage collectors should reclaim all objects that are *dead* at the time of garbage collection. An object is dead at an execution instant if it is not used in future. Since garbage collectors are not able to distinguish between reachable objects that are live and reachable objects that are dead, they conservatively approximate the liveness of an object by its reachability from a set of locations called *root set* (stack locations and registers containing program variables) [14]. As a consequence, many dead objects are left uncollected. This has been confirmed by empirical studies for Haskell [19], Scheme [16] and Java [22,23,24].

Compile time analysis can help in distinguishing reachable objects that are live from reachable objects that are dead. This is done by detecting unused references

[1] Supported by Infosys Technologies Limited, Bangalore, under Infosys Fellowship Award.




(a) Example program.

(b) Memory graph at $\pi_{14}$. Thick edges denote live links. Edges marked $\times$ can be nullified.

Fig. 1. Example Program and its Memory Graph.

to objects. If an object is dead at a program point, none of its references are used by the program beyond that program point. If every unused reference is nullified, then the dead objects may become unreachable and may be claimed by garbage collector.

**Example 1.1** Figure 1(a) shows an example program. The label $\pi$ of an expression $e$ denotes the program point just before the evaluation of $e$. At a given program point, the heap memory can be viewed as a (possibly unconnected) directed acyclic graph called *memory graph*. The locations in the root set form the entry nodes for the memory graph. Figure 1(b) shows the memory graph at $\pi_{14}$. Each *cons* cell is an intermediate *node* in the graph. Elements of basic data types and the 0-ary constructor *Nil* form leaf nodes of the graph. They are assumed to be boxed, i.e. stored in separate heap cells and are accessed through references. The edges in the graph are called *links*.

If we consider the execution of the program starting from $\pi_{14}$, the links in the memory graph that are traversed are shown by thick arrows. These links are *live* at $\pi_{14}$. Links that are not live can be nullified by the compiler by inserting suitable statements. If an object becomes unreachable due to nullification, it can be collected by the garbage collector.

In the figure, the links that can be nullified are shown with a $\times$. Note that a link need not be nullified if nullifying some other link makes it unreachable from the root set. If a node becomes unreachable from the root set as a consequence of nullifying the links, it will be collected during the next invocation of garbage collector. □

In this example, starting at $\pi_{14}$, there is only one execution path. In general, there could be multiple execution paths starting from a program point $\pi$. The liveness information at $\pi$ is a combination of liveness information along every execution path starting at $\pi$.

In this paper, we describe a static analysis for programs written in first-order, eager functional programming languages. The analysis discovers live references at

$$
\begin{array}{lll}
p ::= d_1 \ldots d_n \; e_1 & & \text{— program} \\
d ::= (\textsf{define}\ (f v_1 \;\ldots\; v_n)\ e_1) & & \text{— function definition} \\
e ::= & & \text{— expression} \\
 & \kappa & \text{— constant} \\
 & \mid v & \text{— variable} \\
 & \mid \textsf{Nil} & \text{— primitive constructor} \\
 & \mid (\textsf{cons}\ e_1\ e_2) & \text{— primitive constructor} \\
 & \mid (\textsf{car}\ e_1) & \text{— primitive selector} \\
 & \mid (\textsf{cdr}\ e_1) & \text{— primitive selector} \\
 & \mid (\textsf{pair?}\ e_1) & \text{— primitive tester} \\
 & \mid (\textsf{null?}\ e_1) & \text{— primitive tester} \\
 & \mid (+\ e_1\ e_2) & \text{— generic primitive} \\
 & \mid (\textsf{if}\ e_1\ e_2\ e_3) & \text{— conditional} \\
 & \mid (\textsf{let}\ v_1 \leftarrow e_2\ \textsf{in}\ e_3) & \text{— let binding} \\
 & \mid (f\ e_1\ \ldots\ e_n) & \text{— function application}
\end{array}
$$

Fig. 2. The syntax of our language

every program point, i.e. the references that may be used beyond the program point in any execution of the program. We use context free grammars as a bounded representation for the set of live references. The result of the analysis can be used by the compiler to decide whether a given reference can be nullified at a given program point. Our analysis is context-sensitive yet modular in that a function is analyzed only once.

The rest of the paper is organized as follows: Section 2 describes the language used to explain our analysis along with the basic concepts and notations. The analysis in Section 3 captures the liveness information of a program as a set of equations. The method to solve these equations is given in Section 4. Section 5 describes how the result of the analysis can be used to nullify unused references. Finally, we compare our approach with related work in Section 6 and conclude in Section 7.

## 2 Language, Concepts and Notations

The syntax of our language is described in Figure 2. The language has call-by-value semantics. The argument expressions are evaluated from left to right. We assume that variables in the program are renamed so that the same name is not defined in two different scopes.

For notational convenience, the left link (corresponding to the *car*) of a *cons* cell is denoted by $\mathbf{0}$ and the right link (corresponding to the *cdr*) is denoted by $\mathbf{1}$. We use $e.\mathbf{0}$ to denote the link corresponding to (*car* $e$) for an expression $e$ (assuming $e$ evaluates to a list) and $e.\mathbf{1}$ to denote the link corresponding to (*cdr* $e$). A composition of several *car*s and *cdr*s is represented by a string $\alpha \in \{\mathbf{0}, \mathbf{1}\}^*$. If an expression $e$ evaluates to a *cons* cell then $e.\epsilon$ corresponds to the reference to the *cons* cell.

For an expression $e$, let $[e]$ denote the location in the root set holding the value

of $e$. Given a memory graph, the string $e.\alpha$ describes a path in the memory graph that starts at $[e]$. We call the string $e.\alpha$ an *access expression*, the string $\alpha$ an *access pattern*, and the path traced in the memory graph an *access path*. In Figure 1, the access expression $\mathsf{w}.\mathbf{100}$ represents the access path from $\mathsf{w}$ to the node containing the value 4. Most often, the memory graph being referred to is clear from the context, and therefore we shall use access expressions to refer to access paths. When we use an access path to refer to a link in the memory graph, it denotes the last link in the access path. Thus, $\mathsf{w}.\mathbf{100}$ denotes the link incident on the node containing the value 4. If $\sigma$ denotes a set of access patterns, then $e.\sigma$ is the set of access paths rooted at $[e]$ and corresponding to $\sigma$. i.e.

$$e.\sigma = \{e.\alpha \mid \alpha \in \sigma\}$$

A link in a memory graph is live at a program point $\pi$ if it is used in some path from $\pi$ to the program exit. An access path is defined to be live if its last link is live. In Example 1, the set of live access paths at $\pi_{14}$ is $\{\mathsf{w}.\epsilon, \mathsf{w}.\mathbf{1}, \mathsf{w}.\mathbf{10}, \mathsf{w}.\mathbf{100}, \mathsf{z}.\mathbf{0}, \mathsf{z}.\mathbf{00}\}$. Note that the access paths $\mathsf{z}.\mathbf{0}$ and $\mathsf{z}.\mathbf{00}$ are live at $\pi_{14}$ due to sharing. We do not discover the liveness of such access paths directly. Instead, we assume that an optimizer using our analysis will use alias analysis to discover liveness due to sharing.

The end result of our analysis is the annotation of every expression in the program with a set of access paths rooted at program variables. We call this *liveness environment*, denoted $\mathcal{L}$. This information can be used to insert nullifying statements before expressions.

The symbols $\mathbf{0}$ and $\mathbf{1}$ extend the access patterns of a structure to describe the access patterns of a larger structure. In some situations, we need to create access patterns of a substructure from the access patterns of a larger structure. For this purpose, we extend our alphabet of access patterns to include symbols $\bar{\mathbf{0}}$ and $\bar{\mathbf{1}}$. The following example motivates the need for these symbols.

**Example 2.1** Consider the expression at program point $\pi_1$ in

$$\pi_1 \!:\! (\mathit{let}\ \mathsf{w} \leftarrow \pi_2 \!:\! (\mathit{cons}\ \mathsf{x}\ \mathsf{y})\ \mathit{in}\ \pi_3 \!:\! \cdots)$$

Assuming $\mathcal{L}_{\pi_3} = \{\mathsf{w}.\alpha\}$, we would like to find out which reference of the list $\mathsf{x}$ and $\mathsf{y}$ are live at $\pi_1$. Let $\mathsf{x}.\alpha'$ be live at $\pi_1$. Then, the two possible cases are:

- If $\alpha = \mathbf{1}\beta$ or $\alpha = \epsilon$, no link in the structure rooted at $\mathsf{x}$ is used. We use $\perp$ to denote the access pattern describing such a situation. Thus, $\alpha' = \perp$.
- If $\alpha = \mathbf{0}\beta$ then the link represented by $\mathsf{w}.\alpha$ that is $\mathsf{x}$ rooted and live at $\pi_1$ can be represented by $\mathsf{x}.\beta$. Thus, $\alpha' = \beta$.

This relation between $\alpha$ and $\alpha'$ is expressed by $\alpha' = \bar{\mathbf{0}}\alpha$. $\bar{\mathbf{1}}$ can be interpreted similarly. □

With the inclusion of $\bar{\mathbf{0}}$, $\bar{\mathbf{1}}$ and $\perp$ in the alphabet for access patterns, an access pattern does not directly describe a path in the memory graph. Hence we define a *Canonical Access Pattern* as a string restricted to the alphabet $\{\mathbf{0}, \mathbf{1}\}$. As a special case, $\perp$ is also considered as a canonical access pattern.

We define rules to reduce access patterns to their canonical forms. For access patterns $\alpha_1$ and $\alpha_2$:

(1) $\alpha_1\bar{\mathbf{0}}\alpha_2 \rightarrow \begin{cases} \alpha_1\alpha_2' & \text{if } \alpha_2 \equiv \mathbf{0}\alpha_2' \\ \bot & \text{if } \alpha_2 \equiv \mathbf{1}\alpha_2' \text{ or } \alpha_2 \equiv \epsilon \end{cases}$

(2) $\alpha_1\bar{\mathbf{1}}\alpha_2 \rightarrow \begin{cases} \alpha_1\alpha_2' & \text{if } \alpha_2 \equiv \mathbf{1}\alpha_2' \\ \bot & \text{if } \alpha_2 \equiv \mathbf{0}\alpha_2' \text{ or } \alpha_2 \equiv \epsilon \end{cases}$

(3) $\alpha_1\bot\alpha_2 \rightarrow \bot$

$\alpha \xrightarrow{k} \alpha'$ denotes the reduction of $\alpha$ to $\alpha'$ in $k$ steps, and $\xrightarrow{\star}$ denotes the reflexive and transitive closure of $\rightarrow$. The concatenation $(\cdot)$ of a set of access patterns $\sigma_1$ with $\sigma_2$ is defined as a set containing concatenation of each element in $\sigma_1$ with each element in $\sigma_2$, i.e.

$$\sigma_1 \cdot \sigma_2 = \{\alpha_1\alpha_2 \mid \alpha_1 \in \sigma_1, \alpha_2 \in \sigma_2\}$$

## 3 Computing Liveness Environments

Let $\sigma$ be the set of access patterns specifying the liveness of the result of evaluating $e$. Let $\mathcal{L}$ be the liveness environment after the evaluation of $e$. Then the liveness environment before the computation of $e$ is discovered by propagating $\sigma$ backwards through the body of $e$. This is achieved by defining an environment transformer for $e$, denoted $\mathcal{XE}$.

Since $e$ may contain applications of primitive operations and user defined functions, we also need transfer functions that propagate $\sigma$ from the result of the application to the arguments. These functions are denoted by $\mathcal{XP}$ and $\mathcal{XF}$. While $\mathcal{XP}$ is given directly based on the semantics of the primitive, $\mathcal{XF}$ is inferred from the body of a function.

### *3.1 Computing $\mathcal{XE}$*

For an expression $e$ at program point $\pi$, $\mathcal{XE}(e, \sigma, \mathcal{L})$ computes liveness environment at $\pi$ where $\sigma$ is the set of access patterns specifying the liveness of the result of evaluating $e$ and $\mathcal{L}$ is the liveness environment after the evaluation of $e$. Additionally, as a side effect, the program point $\pi$ is annotated with the value computed. However, we do not show this explicitly to avoid clutter. The computation of $\mathcal{XE}(e, \sigma, \mathcal{L})$ is as follows.

(4) $\mathcal{XE}(\kappa, \sigma, \mathcal{L}) = \mathcal{L}$

(5) $\mathcal{XE}(v, \sigma, \mathcal{L}) = \mathcal{L} \cup v.\sigma$

(6) $\mathcal{XE}((P\ e_1\ e_2), \sigma, \mathcal{L}) = \textsf{\textit{let}}\ \mathcal{L}' \leftarrow \mathcal{XE}(e_2, \mathcal{XP}_P^2(\sigma), \mathcal{L})\ \textsf{\textit{in}}$
$\quad\quad \mathcal{XE}(e_1, \mathcal{XP}_P^1(\sigma), \mathcal{L}')$
where $P$ is one of *cons*, $+$

(7) $\mathcal{XE}((P\ e_1), \sigma, \mathcal{L}) = \mathcal{XE}(e_1, \mathcal{XP}_P^1(\sigma), \mathcal{L})$
where $P$ is one of *car*, *cdr*, *null?*, *pair?*

(8) $\mathcal{XE}((\textsf{\textit{if}}\ e_1\ e_2\ e_3), \sigma, \mathcal{L}) = \textsf{\textit{let}}\ \mathcal{L}' \leftarrow \mathcal{XE}(e_3, \sigma, \mathcal{L})\ \textsf{\textit{in}}$
$\quad\quad \textsf{\textit{let}}\ \mathcal{L}'' \leftarrow \mathcal{XE}(e_2, \sigma, \mathcal{L})\ \textsf{\textit{in}}$
$\quad\quad\quad \mathcal{XE}(e_1, \{\epsilon\}, \mathcal{L}' \cup \mathcal{L}'')$

(9) $\mathcal{XE}((\textsf{\textit{let}}\ v_1 \leftarrow e_1\ \textsf{\textit{in}}\ e_2), \sigma, \mathcal{L}) = \textsf{\textit{let}}\ \mathcal{L}' \leftarrow \mathcal{XE}(e_2, \sigma, \mathcal{L})\ \textsf{\textit{in}}$

$$\mathcal{XE}(e_1, \sigma', \mathcal{L}' - v_1.\sigma')$$
$$\text{where } \sigma' = \{\alpha \mid v_1.\alpha \in \mathcal{L}'\}$$

(10) $$\mathcal{XE}((f\ e_1 \ldots e_n), \sigma, \mathcal{L}) = \textit{let } \mathcal{L}_1 \leftarrow \mathcal{XE}(e_n, \mathcal{XF}_f^n(\sigma), \mathcal{L}) \textit{ in}$$
$$\vdots$$
$$\textit{let } \mathcal{L}_{n-1} \leftarrow \mathcal{XE}(e_2, \mathcal{XF}_f^2(\sigma), \mathcal{L}_{n-2}) \textit{ in}$$
$$\mathcal{XE}(e_1, \mathcal{XF}_f^1(\sigma), \mathcal{L}_{n-1})$$

We explain the definition of $\mathcal{XE}$ for the *if* expression. Since the value of the conditional expression $e_1$ is boolean and this value is used, the liveness access pattern with respect to which $e_1$ is computed is $\{\epsilon\}$. Further, since it is not possible to statically determine whether $e_2$ or $e_3$ will be executed, the liveness environment with respect to which $e_1$ is computed is the union of the liveness environments arising out of $e_2$ and $e_3$.

### *3.2 Computing $\mathcal{XP}$ and $\mathcal{XF}$*

If $\sigma$ is the set of access patterns specifying the liveness of the result of evaluating $(P\ e_1 \ldots e_n)$, where $P$ is a primitive, then $\mathcal{XP}_P^i(\sigma)$ gives the set of access patterns specifying the liveness of $e_i$. We describe the transfer functions for the primitives in our language: *car*, *cdr*, *cons*, *null?*, *pair?* and +. The 0-ary constructor *Nil* does not accept any argument and is ignored.

Assume that the live access pattern for the result of the expression (*car* $e$) is $\alpha$. Then, the link that is denoted by the path labeled $\alpha$ starting from location [(*car* $e$)] can also be denoted by a path $\mathbf{0}\alpha$ starting from location $[e]$. We can extend the same reasoning for set of access patterns ($\sigma$) of result, i.e. every pattern in the set is prefixed by $\mathbf{0}$ to give live access pattern of $e$. Also, since the cell corresponding to $e$ is used to find the value of *car*, we need to add $\epsilon$ to the live access patterns of $e$. Reasoning about (*cdr* $e$) similarly, we have

(11) $\mathcal{XP}_{\textit{car}}^1(\sigma) = \{\epsilon\} \cup \{\mathbf{0}\} \cdot \sigma, \quad \mathcal{XP}_{\textit{cdr}}^1(\sigma) = \{\epsilon\} \cup \{\mathbf{1}\} \cdot \sigma$

As seen in Example 2.1, an access pattern of $\alpha$ for result of *cons* translates to an access pattern of $\bar{\mathbf{0}}\alpha$ for its first argument, and $\bar{\mathbf{1}}\alpha$ for its second argument. Since *cons* does not read its arguments, the access patterns of the arguments do not contain $\epsilon$.

(12) $\mathcal{XP}_{\textit{cons}}^1(\sigma) = \{\bar{\mathbf{0}}\} \cdot \sigma, \quad \mathcal{XP}_{\textit{cons}}^2(\sigma) = \{\bar{\mathbf{1}}\} \cdot \sigma$

Since the remaining primitives read only the value of the arguments, the set of live access patterns of the arguments is $\{\epsilon\}$.

(13) $\mathcal{XP}_{\textit{null?}}^1(\sigma) = \{\epsilon\}, \quad \mathcal{XP}_{\textit{pair?}}^1(\sigma) = \{\epsilon\}, \quad \mathcal{XP}_{+}^1(\sigma) = \{\epsilon\}, \quad \mathcal{XP}_{+}^2(\sigma) = \{\epsilon\}$

We now consider the transfer function for a user defined function $f$. If $\sigma$ is the set of access patterns specifying the liveness of the result of evaluating $(f\ e_1 \ldots e_n)$, then $\mathcal{XF}_f^i(\sigma)$ gives the set of access patterns specifying the liveness of $e_i$. Let $f$ be defined as:

$$(\textit{define}\ (f\ v_1\ \ldots\ v_n)\ \ \pi\!: e)$$

Assume that $\sigma$ is the live access pattern for the result of $f$. Then, $\sigma$ is also the live

access pattern for $e$. $\mathcal{XE}(e, \sigma, \emptyset)$ computes live access patterns for $v_i$ $(1 \leq i \leq n)$ at $\pi$. Thus, the transfer function for the $i^{th}$ argument of $f$ is given by:

(14) $\mathcal{XF}_f^i(\sigma) = \{\alpha \mid v_i.\alpha \in \mathcal{XE}(e, \sigma, \emptyset)\} \quad 1 \leq i \leq n$

The following example illustrates our analysis.

**Example 3.1** Consider the program in Figure 1. To compute the transfer functions for app, we compute the environment transformer $\mathcal{XE}(e, \sigma, \emptyset)$ in terms of a variable $\sigma$. Here $e$ is the body of app. The value of the liveness environment at each point in the body of app is shown in Appendix **??**. From the liveness information at $\pi_1$ we get:

$$\mathcal{XF}^1_{\mathsf{app}}(\sigma) = \{\epsilon\} \cup \{\mathbf{0\bar{0}}\} \cdot \sigma \ \cup \ \{\mathbf{1}\} \cdot \mathcal{XF}^1_{\mathsf{app}}(\{\mathbf{\bar{1}}\} \cdot \sigma)$$
$$\mathcal{XF}^2_{\mathsf{app}}(\sigma) = \sigma \cup \mathcal{XF}^2_{\mathsf{app}}(\{\mathbf{\bar{1}}\} \cdot \sigma)$$

Let $e_{\mathsf{pgm}}$ represent the entire program being analyzed and $\sigma_{\mathsf{pgm}}$ be the set of access patterns describing the liveness of the result. Then, the liveness environment at various points in the $e_{\mathsf{pgm}}$ can be computed as $\mathcal{XE}(e_{\mathsf{pgm}}, \sigma_{\mathsf{pgm}}, \emptyset)$. The liveness environments at $\pi_{14}$ and $\pi_{12}$ are as follows:

$$\mathcal{L}_{\pi_{14}} = \{\ \mathsf{w}.(\{\epsilon, \mathbf{1}, \mathbf{10}\} \cup \{\mathbf{100}\} \cdot \sigma_{\mathsf{pgm}})\ \}$$
$$\mathcal{L}_{\pi_{12}} = \left\{ \begin{array}{l} \mathsf{y}.\mathcal{XF}^1_{\mathsf{app}}(\{\epsilon, \mathbf{1}, \mathbf{10}\} \cup \{\mathbf{100}\} \cdot \sigma_{\mathsf{pgm}}), \\ \mathsf{z}.\mathcal{XF}^2_{\mathsf{app}}(\{\epsilon, \mathbf{1}, \mathbf{10}\} \cup \{\mathbf{100}\} \cdot \sigma_{\mathsf{pgm}}) \end{array} \right\}$$

□

We assume that the entire result of the program is needed, i.e., $\sigma_{\mathsf{pgm}}$ is $\{\mathbf{0}, \mathbf{1}\}*$.

# 4 Solving the Equations for $\mathcal{XF}$

In general, the equations defining the transfer functions $\mathcal{XF}$ will be recursive. To solve such equations, we start by guessing that the solution will be of the form:

(15) $\mathcal{XF}_f^i(\sigma) = \mathcal{I}_f^i \cup \mathcal{D}_f^i \cdot \sigma,$

where $\mathcal{I}_f^i$ and $\mathcal{D}_f^i$ are sets of strings over the alphabet $\{\mathbf{0}, \mathbf{1}, \mathbf{\bar{0}}, \mathbf{\bar{1}}\}$. The intuition behind this form of solution is as follows: The function $f$ can use its argument locally and/or copy a part of it to the return value being computed. $\mathcal{I}_f^i$ is the live access pattern of $i^{th}$ argument due to local use in $f$. $\mathcal{D}_f^i$ is a sort of selector that selects the liveness pattern corresponding to the $i^{th}$ argument of $f$ from $\sigma$, the liveness pattern of the return value.

If we substitute the guessed form of $\mathcal{XF}_f^i$ in the equations describing it and equate the terms containing $\sigma$ and the terms without $\sigma$, we get the equations for $\mathcal{I}_f^i$ and $\mathcal{D}_f^i$. This is illustrated in the following example.

**Example 4.1** Consider the equation for $\mathcal{XF}^1_{\mathsf{app}}(\sigma)$ from Example 3.1:

$$\mathcal{XF}^1_{\mathsf{app}}(\sigma) = \{\epsilon\} \cup \{\mathbf{0\bar{0}}\} \cdot \sigma \cup \{\mathbf{1}\} \cdot \mathcal{XF}^1_{\mathsf{app}}(\{\mathbf{\bar{1}}\} \cdot \sigma)$$

Decomposing both sides of the equation, and rearranging the RHS gives:

$$\begin{aligned} \mathcal{I}^1_{\mathsf{app}} \cup \mathcal{D}^1_{\mathsf{app}} \cdot \sigma &= \{\epsilon\} \cup \{\mathbf{0\bar{0}}\} \cdot \sigma \cup \{\mathbf{1}\} \cdot (\mathcal{I}^1_{\mathsf{app}} \cup \ \mathcal{D}^1_{\mathsf{app}} \cdot \{\mathbf{\bar{1}}\} \cdot \sigma) \\ &= \{\epsilon\} \cup \{\mathbf{1}\} \cdot \mathcal{I}^1_{\mathsf{app}} \cup \{\mathbf{0\bar{0}}\} \cdot \sigma \cup \{\mathbf{1}\} \cdot \mathcal{D}^1_{\mathsf{app}} \cdot \{\mathbf{\bar{1}}\} \cdot \sigma \end{aligned}$$

Separating the parts that are $\sigma$ dependent and the parts that are $\sigma$ independent, and equating them separately, we get:

$$\begin{aligned}\mathcal{I}^1_{\mathsf{app}} &= \{\epsilon\} \cup \{\mathbf{1}\} \cdot \mathcal{I}^1_{\mathsf{app}} \\ \mathcal{D}^1_{\mathsf{app}} \cdot \sigma &= \{\mathbf{0\bar{0}}\} \cdot \sigma \cup \{\mathbf{1}\} \cdot \mathcal{D}^1_{\mathsf{app}} \cdot \{\mathbf{\bar{1}}\}\sigma \\ &= (\{\mathbf{0\bar{0}}\} \cup \{\mathbf{1}\} \cdot \mathcal{D}^1_{\mathsf{app}} \cdot \{\mathbf{\bar{1}}\}) \cdot \sigma\end{aligned}$$

As the equations hold for any general $\sigma$, we can simplify them to:

$$\mathcal{I}^1_{\mathsf{app}} = \{\epsilon\} \cup \{\mathbf{1}\} \cdot \mathcal{I}^1_{\mathsf{app}} \quad \text{and} \quad \mathcal{D}^1_{\mathsf{app}} = \{\mathbf{0\bar{0}}\} \cup \{\mathbf{1}\} \cdot \mathcal{D}^1_{\mathsf{app}} \cdot \{\mathbf{\bar{1}}\}$$

Similarly, from the equation describing $\mathcal{XF}^2_{\mathsf{app}}(\sigma)$, we get:

$$\mathcal{I}^2_{\mathsf{app}} = \mathcal{I}^2_{\mathsf{app}} \quad \text{and} \quad \mathcal{D}^2_{\mathsf{app}} = \{\epsilon\} \cup \mathcal{D}^2_{\mathsf{app}} \cdot \{\mathbf{\bar{1}}\}$$

The liveness environment at $\pi_{12}$ and $\pi_{14}$ in terms $\mathcal{I}_{\mathsf{app}}$ and $\mathcal{D}_{\mathsf{app}}$ are:

$$\mathcal{L}_{\pi_{14}} = \{\ \mathsf{w}.\{\mathbf{100}\} \cdot \sigma_{\mathsf{pgm}}\ \}$$

$$\mathcal{L}_{\pi_{12}} = \left\{\begin{array}{l}\mathsf{y}.(\mathcal{I}^1_{\mathsf{app}}\ \cup\ \mathcal{D}^1_{\mathsf{app}} \cdot (\{\epsilon, \mathbf{1}, \mathbf{10}\} \cup \{\mathbf{100}\} \cdot \sigma_{\mathsf{pgm}})), \\ \mathsf{z}.(\mathcal{I}^2_{\mathsf{app}}\ \cup\ \mathcal{D}^2_{\mathsf{app}} \cdot (\{\epsilon, \mathbf{1}, \mathbf{10}\} \cup \{\mathbf{100}\} \cdot \sigma_{\mathsf{pgm}}))\end{array}\right\}$$

Solving for $\mathcal{I}_{\mathsf{app}}$ and $\mathcal{D}_{\mathsf{app}}$ gives us the desired liveness environments at these program points. □

### *4.1 Representing Liveness by Context Free Grammars*

The values of $\mathcal{I}$ and $\mathcal{D}$ variables of a transfer function are sets of strings over the alphabet $\{\mathbf{0}, \mathbf{1}, \mathbf{\bar{0}}, \mathbf{\bar{1}}\}$. We use context free grammars (CFG) to describe these sets. The set of terminal symbols of the CFG is $\{\mathbf{0}, \mathbf{1}, \mathbf{\bar{0}}, \mathbf{\bar{1}}\}$. Non-terminals and associated rules are constructed as illustrated in Examples 4.2 and 4.3.

**Example 4.2** Consider the following constraint from Example 4.1:

$$\mathcal{I}^1_{\mathsf{app}} = \{\epsilon\} \cup \{\mathbf{1}\} \cdot \mathcal{I}^1_{\mathsf{app}}$$

We add non-terminal $\langle\mathcal{I}^1_{\mathsf{app}}\rangle$ and the productions with right hand sides directly derived from the constraints:

$$\langle\mathcal{I}^1_{\mathsf{app}}\rangle \to \epsilon \mid \mathbf{1}\langle\mathcal{I}^1_{\mathsf{app}}\rangle$$

The productions generated from other constraints of Example 4.1 are:

$$\begin{aligned}\langle\mathcal{D}^1_{\mathsf{app}}\rangle &\to \mathbf{0\bar{0}} \mid \mathbf{1}\langle\mathcal{D}^1_{\mathsf{app}}\rangle\mathbf{\bar{1}} \\ \langle\mathcal{I}^2_{\mathsf{app}}\rangle &\to \langle\mathcal{I}^2_{\mathsf{app}}\rangle \\ \langle\mathcal{D}^2_{\mathsf{app}}\rangle &\to \epsilon \mid \langle\mathcal{D}^2_{\mathsf{app}}\rangle\mathbf{\bar{1}}\end{aligned}$$

These productions describe the transfer functions of app. □

The liveness environment at each program point can be represented as a CFG with a start symbol for every variable. To do so, the analysis starts with $\langle S_{\mathsf{pgm}}\rangle$, the non-terminal describing the liveness pattern of the result of the program, $\sigma_{\mathsf{pgm}}$. The productions for $\langle S_{\mathsf{pgm}}\rangle$ are:

$$\langle S_{\mathsf{pgm}}\rangle \to \epsilon \mid \mathbf{0}\langle S_{\mathsf{pgm}}\rangle \mid \mathbf{1}\langle S_{\mathsf{pgm}}\rangle$$

**Example 4.3** Let $S^{\mathsf{v}}_{\pi}$ denote the non-terminal generating liveness access patterns associated with a variable v at program point $\pi$. For the program of Figure 1:

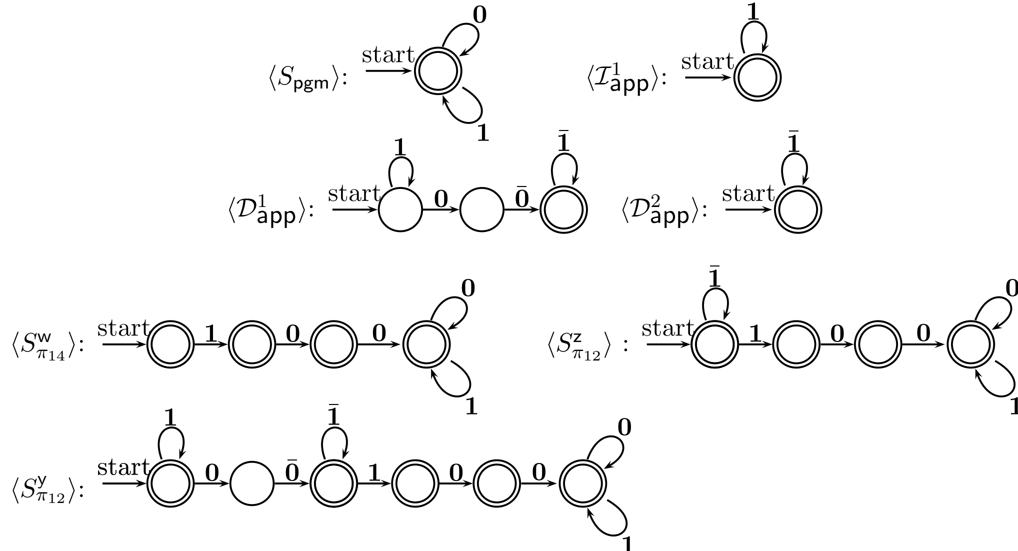
$\langle S^{\mathsf{w}}_{\pi_{14}}\rangle \rightarrow \epsilon \mid \mathbf{1} \mid \mathbf{10} \mid \mathbf{100}\langle S_{\mathsf{pgm}}\rangle$
$\langle S^{\mathsf{z}}_{\pi_{12}}\rangle \rightarrow \langle \mathcal{I}^2_{\mathsf{app}}\rangle \mid \langle \mathcal{D}^2_{\mathsf{app}}\rangle \mid \langle \mathcal{D}^2_{\mathsf{app}}\rangle\mathbf{1} \mid \langle \mathcal{D}^2_{\mathsf{app}}\rangle\mathbf{10} \mid \langle \mathcal{D}^2_{\mathsf{app}}\rangle\mathbf{100}\langle S_{\mathsf{pgm}}\rangle$
$\langle S^{\mathsf{y}}_{\pi_{12}}\rangle \rightarrow \langle \mathcal{I}^1_{\mathsf{app}}\rangle \mid \langle \mathcal{D}^1_{\mathsf{app}}\rangle \mid \langle \mathcal{D}^1_{\mathsf{app}}\rangle\mathbf{1} \mid \langle \mathcal{D}^1_{\mathsf{app}}\rangle\mathbf{10} \mid \langle \mathcal{D}^1_{\mathsf{app}}\rangle\mathbf{100}\langle S_{\mathsf{pgm}}\rangle$ □

The access patterns in the access paths used for nullification are in canonical form but the access patterns described by the CFGs resulting out of our analysis are not. It is not obvious how to check the membership of a canonical access pattern in such CFGs. To solve this problem, we need equivalent CFGs such that if $\alpha$ belongs to an original CFG and $\alpha \xrightarrow{\star} \beta$, where $\beta$ is in canonical form, then $\beta$ belongs to the corresponding new CFG. Directly converting the reduction rules (Equations (1, 2, 3)) into productions and adding it to the grammar results in *unrestricted grammar* [11]. To simplify the problem, we approximate original CFGs by non-deterministic finite automata (NFAs) and eliminate $\bar{\mathbf{0}}$ and $\bar{\mathbf{1}}$ from the NFAs.

### *4.2 Approximating CFGs using NFAs*

The conversion of a CFG $G$ to an approximate NFA $\mathbf{N}$ should be safe in that the language accepted by $\mathbf{N}$ should be a superset of the language accepted by $G$. We use the algorithm described by Mohri and Nederhof [18]. The algorithm transforms a CFG to a restricted form called *strongly regular* CFG which can be converted easily to a finite automaton.

**Example 4.4** We show the approximate NFAs for each of the non-terminals in Example 4.2 and Example 4.3.




Note that there is no automaton for $\langle \mathcal{I}^2_{\mathsf{app}}\rangle$. This is because the least solution of the equation $\langle \mathcal{I}^2_{\mathsf{app}}\rangle \rightarrow \langle \mathcal{I}^2_{\mathsf{app}}\rangle$ is $\emptyset$. Also, the language accepted by the automaton for $\mathcal{D}^1_{\mathsf{app}}$ is approximate as it does not ensure that there is an equal number of $\mathbf{1}$ and $\bar{\mathbf{1}}$ in the strings generated by rules for $\langle \mathcal{D}^1_{\mathsf{app}}\rangle$. □

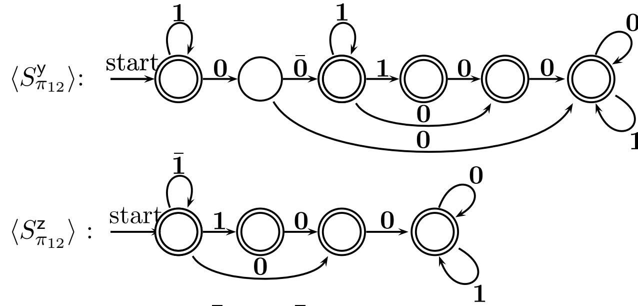
### *4.3 Eliminating $\bar{\mathbf{0}}$ and $\bar{\mathbf{1}}$ from NFA*

We now describe how to convert an NFA with transitions on symbols $\bar{\mathbf{0}}$ and $\bar{\mathbf{1}}$ to an equivalent NFA without any transitions on these symbols.
**Input:** An NFA $\overline{\mathbf{N}}$ with underlying alphabet $\{\mathbf{0}, \mathbf{1}, \bar{\mathbf{0}}, \bar{\mathbf{1}}\}$ accepting a set of access patterns
**Output:** An NFA $\mathbf{N}$ with underlying alphabet $\{\mathbf{0}, \mathbf{1}\}$ accepting the equivalent set of canonical access patterns
**Steps:**

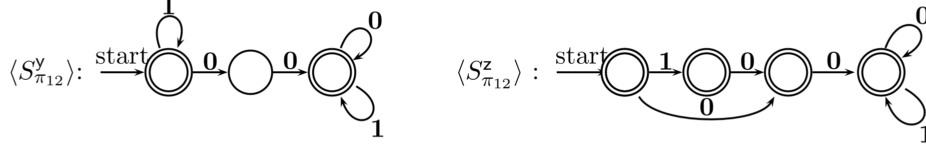
$i \leftarrow 0$
$\mathbf{N}_0 \leftarrow$ Equivalent NFA of $\overline{\mathbf{N}}$ without $\epsilon$-moves [11]
**do** {
  $\mathbf{N}'_{i+1} \leftarrow \mathbf{N}_i$
  foreach state $q$ in $\mathbf{N}_i$ such that $q$ has an incoming edge from $q'$
  with label $\bar{\mathbf{0}}$ and outgoing edge to $q''$ with label $\mathbf{0}$ {
    /$\star$ bypass $\bar{\mathbf{0}}\mathbf{0}$ using $\epsilon$ $\star$/
    add an edge in $\mathbf{N}'_{i+1}$ from $q'$ to $q''$ with label $\epsilon$.
  }
  foreach state $q$ in $\mathbf{N}_i$ such that $q$ has an incoming edge from $q'$
  with label $\bar{\mathbf{1}}$ and outgoing edge to $q''$ with label $\mathbf{1}$ {
    /$\star$ bypass $\bar{\mathbf{1}}\mathbf{1}$ using $\epsilon$ $\star$/
    add an edge in $\mathbf{N}'_{i+1}$ from $q'$ to $q''$ with label $\epsilon$.
  }
  $\mathbf{N}_{i+1} \leftarrow$ Equivalent NFA of $\mathbf{N}'_{i+1}$ without $\epsilon$-moves
  $i \leftarrow i + 1$
} **while** $(\mathbf{N}_i \neq \mathbf{N}_{i-1})$
$\mathbf{N} \leftarrow \mathbf{N}_i$
delete all edges with label $\bar{\mathbf{0}}$ or $\bar{\mathbf{1}}$ in $\mathbf{N}$.

The algorithm repeatedly introduces $\epsilon$ edges to bypass a pair of consecutive edges labeled $\bar{\mathbf{0}}\mathbf{0}$ or $\bar{\mathbf{1}}\mathbf{1}$. The process is continued till a fixed point is reached. When the fixed point is reached, the resulting NFA contains the canonical access patterns corresponding to all the access patterns in the original NFA. The access patterns not in canonical form are deleted by removing edges labeled $\bar{\mathbf{0}}$ and $\bar{\mathbf{1}}$. Note that by our reduction rules if $\alpha$ is accepted by $\overline{\mathbf{N}}$ and $\alpha \xrightarrow{\star} \bot$, then $\bot$ should be accepted by $\mathbf{N}$, However, $\mathbf{N}$ returned by our algorithm does not accept $\bot$. This is not a problem because the access patterns which are tested for membership against $\mathbf{N}$ do not include $\bot$ as well.

**Example 4.5** We show the elimination of $\bar{\mathbf{0}}$ and $\bar{\mathbf{1}}$ for the automata for $\langle S^{\mathsf{y}}_{\pi_{12}} \rangle$ and $\langle S^{\mathsf{z}}_{\pi_{12}} \rangle$. The automaton for $\langle S^{\mathsf{w}}_{\pi_{14}} \rangle$ remains unchanged as it does not contain transitions on $\bar{\mathbf{0}}$ and $\bar{\mathbf{1}}$. The automata at the termination of the loop in the algorithm are:




Eliminating the edges labeled $\bar{\mathbf{0}}$ and $\bar{\mathbf{1}}$, and removing the dead states gives:




The language accepted by these automata represent the live access paths corresponding to y and z at $\pi_{12}$. □

We now prove the termination and correctness of our algorithm.

*Termination*

Termination of the algorithm follows from the fact that every iteration of **do-while** loop adds new edges to the NFA, while old edges are not deleted. Since no new states are added to NFA, only a fixed number of edges can be added before we reach a fix point.

*Correctness*

The sequence of obtaining $\mathbf{N}$ from $\overline{\mathbf{N}}$ can be viewed as follows, with $\mathbf{N}_m$ denoting the NFA at the termination of while loop:

$$\overline{\mathbf{N}} \xrightarrow[\text{of } \epsilon\text{-edges}]{\text{deletion}} \mathbf{N}_0 \xrightarrow[\text{of } \epsilon\text{-edges}]{\text{addition}} \mathbf{N}'_1 \xrightarrow[\text{of } \epsilon\text{-edges}]{\text{deletion}} \mathbf{N}_1 \xrightarrow[\text{of } \epsilon\text{-edges}]{\text{addition}} \cdots \xrightarrow[\text{of } \epsilon\text{-edges}]{\text{addition}} \mathbf{N}'_i \xrightarrow[\text{of } \epsilon\text{-edges}]{\text{deletion}} \mathbf{N}_i \cdots \xrightarrow[\text{of } \epsilon\text{-edges}]{\text{deletion}} \mathbf{N}_m$$

$$\mathbf{N}_m \xrightarrow[\bar{\mathbf{0}},\ \bar{\mathbf{1}} \text{ edges}]{\text{deletion of}} \mathbf{N}$$

Then, the languages accepted by these NFAs have the following relation:

$$L(\overline{\mathbf{N}}) = L(\mathbf{N}_0) \subseteq L(\mathbf{N}'_1) = L(\mathbf{N}_1) \subseteq \cdots \subseteq L(\mathbf{N}'_i) = L(\mathbf{N}_i) \subseteq \cdots = L(\mathbf{N}_m)$$

$$L(\mathbf{N}) \subseteq L(\mathbf{N}_m)$$

We first prove that the addition of $\epsilon$-edges in the while loop does not add any new information, i.e. any access pattern accepted by the NFA after the addition of $\epsilon$-edges is a reduced version of some access pattern existing in the NFA before the addition of $\epsilon$-edges.

**Lemma 4.6** *for $i > 0$, if $\alpha \in L(\mathbf{N}_i)$ then there exists $\alpha' \in L(\mathbf{N}_{i-1})$ such that $\alpha' \xrightarrow{\star} \alpha$.*

**Proof.** As $L(\mathbf{N}_i) = L(\mathbf{N}'_i)$, we have $\alpha \in L(\mathbf{N}'_i)$. Only difference between $\mathbf{N}'_i$ and $\mathbf{N}_{i-1}$ is that $\mathbf{N}'_i$ contains some extra $\epsilon$-edges. Thus, any $\epsilon$-edge free path in $\mathbf{N}'_i$ is also in $\mathbf{N}_{i-1}$. Consider a path $p$ in $\mathbf{N}'_i$ that accepts $\alpha$. Assume the number of $\epsilon$

edges in $p$ is $k$. The proof is by induction on $k$.
($BASE$) $k = 0$, i.e. $p$ does not contains any $\epsilon$-edge: As the path $p$ is $\epsilon$-edge free, it must be present in $\mathbf{N}_{i-1}$. Thus, $\mathbf{N}_{i-1}$ also accepts $\alpha$. $\alpha \xrightarrow{\star} \alpha$.
($HYPOTHESIS$) For any $\alpha \in L(\mathbf{N}_i)$ with accepting path $p$ having less than $k$ $\epsilon$-edges there exists $\alpha' \in L(\mathbf{N}_{i-1})$ such that $\alpha' \xrightarrow{\star} \alpha$.
($INDUCTION$) $p$ contains $k$ $\epsilon$-edges $e_1, \ldots, e_k$: Assume $e_1$ connects states $q'$ and $q''$ in $\mathbf{N}'_i$. By construction, there exists a state $q$ in $\mathbf{N}'_i$ such that there is an edge $e'_1$ from $q'$ to $q$ with label $\bar{\mathbf{0}}(\bar{\mathbf{1}})$ and an edge $e''_1$ from $q$ to $q''$ with label $\mathbf{0}(\mathbf{1})$ in $\mathbf{N}'_i$. Replace $e_1$ by $e'_1 e''_1$ in $p$ to get a new path $p''$ in $\mathbf{N}'_i$. Let $\alpha''$ be the access pattern accepted by $p''$. Clearly, $\alpha'' \xrightarrow{1} \alpha$. Since $p''$ has $k-1$ $\epsilon$-edges, $\alpha''$ is accepted by $\mathbf{N}'_i$ along a path ($p''$) that has less than $k$ $\epsilon$-edges. By induction hypothesis, we have $\alpha' \in L(\mathbf{N}_{i-1})$ such that $\alpha' \xrightarrow{\star} \alpha''$. This along with $\alpha'' \xrightarrow{1} \alpha$ gives $\alpha' \xrightarrow{\star} \alpha$. □

**Corollary 4.7** *for each $\alpha \in L(\mathbf{N}_m)$, there exists $\alpha' \in L(\overline{\mathbf{N}})$ such that $\alpha' \xrightarrow{\star} \alpha$.*

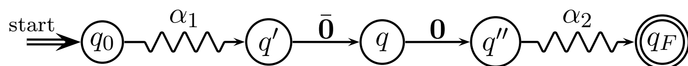
**Proof.** The proof is by induction on $m$, and using Lemma 4.6. □

The following lemma shows that the the language accepted by $\mathbf{N}_m$ is closed with respect to reduction of access patterns.

**Lemma 4.8** *For $\alpha \in L(\mathbf{N}_m)$, if $\alpha \xrightarrow{\star} \alpha'$ and $\alpha' \neq \bot$, then $\alpha' \in L(\mathbf{N}_m)$.*

**Proof.** Assume $\alpha \xrightarrow{k} \alpha'$. The Proof is by induction on $k$, number of steps in reduction.
($BASE$) case $k = 0$ is trivial as $\alpha \xrightarrow{0} \alpha$.
($HYPOTHESIS$) Assume that for $\alpha \in L(\mathbf{N}_m)$, if $\alpha \xrightarrow{k-1} \alpha'$, then $\alpha' \in L(\mathbf{N}_m)$.
($INDUCTION$) $\alpha \in L(\mathbf{N}_m)$, $\alpha \xrightarrow{k} \alpha'$. There exists $\alpha''$ such that: $\alpha \xrightarrow{k-1} \alpha'' \xrightarrow{1} \alpha'$. By induction hypothesis, we have $\alpha'' \in L(\mathbf{N}_m)$.

For $\alpha'' \xrightarrow{1} \alpha'$ to hold we must have $\alpha'' = \alpha_1 \bar{\mathbf{0}}\mathbf{0} \alpha_2$ and $\alpha' = \alpha_1\alpha_2$, or $\alpha'' = \alpha_1 \bar{\mathbf{1}}\mathbf{1}\alpha_2$ and $\alpha' = \alpha_1\alpha_2$. Consider the case when $\alpha'' = \alpha_1 \bar{\mathbf{0}}\mathbf{0}\alpha_2$. Any path in $\mathbf{N}_m$ accepting $\alpha''$ must have the following structure (The states shown separately may not necessarily be different):




As $\mathbf{N}_m$ is the fixed point NFA for the iteration process described in the algorithm, adding an $\epsilon$-edge between states $q'$ and $q''$ will not change the language accepted by $\mathbf{N}_m$. But, the access pattern accepted after adding an $\epsilon$-edge is $\alpha_1\alpha_2 = \alpha'$. Thus, $\alpha' \in L(\mathbf{N}_m)$. The case when $\alpha'' = \alpha_1\bar{\mathbf{1}}\mathbf{1}\alpha_2$ is identical. □

**Corollary 4.9** *For $\alpha \in L(\overline{\mathbf{N}})$, if $\alpha \xrightarrow{\star} \alpha'$ and $\alpha' \neq \bot$, then $\alpha' \in L(\mathbf{N}_m)$.*

**Proof.** $L(\overline{\mathbf{N}}) \subseteq L(\mathbf{N}_m) \Rightarrow \alpha \in L(\mathbf{N}_m)$. The proof follows from Lemma 4.8. □

The following theorem asserts the equivalence of $\overline{\mathbf{N}}$ and $\mathbf{N}$ with respect to the equivalence of access patterns, i.e. every access pattern in $\overline{\mathbf{N}}$ has an equivalent canonical access pattern in $\mathbf{N}$, and for every canonical access pattern in $\mathbf{N}$, there exists an equivalent access pattern in $\overline{\mathbf{N}}$.

**Theorem 4.10** *Let $\overline{\mathbf{N}}$ be an NFA with underlying alphabet $\{\mathbf{0}, \mathbf{1}, \bar{\mathbf{0}}, \bar{\mathbf{1}}\}$. Let NFA $\mathbf{N}$ be the NFA with underlying alphabet $\{\mathbf{0}, \mathbf{1}\}$ returned by the algorithm. Then,*

(i) *if $\alpha \in L(\overline{\mathbf{N}})$, $\beta$ is a canonical access pattern such that $\alpha \xrightarrow{\star} \beta$ and $\beta \neq \bot$, then $\beta \in L(\mathbf{N})$.*

(ii) *if $\beta \in L(\mathbf{N})$ then there exists an access pattern $\alpha \in L(\overline{\mathbf{N}})$ such that $\alpha \xrightarrow{\star} \beta$.*

**Proof.**

(i) From Corollary 4.9: $\alpha \in L(\overline{\mathbf{N}}), \alpha \xrightarrow{\star} \beta$ and $\beta \neq \bot \Rightarrow \beta \in L(\mathbf{N}_m)$. As $\beta$ is in canonical form, the path accepting $\beta$ in $\mathbf{N}_m$ consists of edges labeled $\mathbf{0}$ and $\mathbf{1}$ only. The same path exists in $\mathbf{N}$. Thus $\mathbf{N}$ also accepts $\beta \Rightarrow \beta \in L(\mathbf{N})$.

(ii) $L(\mathbf{N}) \subseteq L(\mathbf{N}_m) \Rightarrow \beta \in L(\mathbf{N}_m)$. Using Corollary 4.7, there exists $\alpha \in L(\overline{\mathbf{N}})$ such that $\alpha \xrightarrow{\star} \beta$.

□

## 5 An Application of Liveness Analysis

The result of liveness analysis can be used to decide whether a given access path $v.\alpha$ can be nullified at a given program point $\pi$. Let the link corresponding to $v.\alpha$ in the memory graph be $l$. A naive approach is to nullify $v.\alpha$ if it does not belongs to the liveness environment at $\pi$. However, the approach is not safe because of two reasons: (a) The link $l$ may be used beyond $\pi$ through an alias, and may therefore be live. (b) a link $l'$ in the access path from the root variable $v$ to $l$ may have been created along one execution path but not along another. Since the nullification of $v.\alpha$ requires the link $l'$ to be dereferenced, a run time exception may occur.

To solve the first problem, we need an alias analysis phase to detect sharing of links among access paths. A link in the memory graph can be nullified at $\pi$ if none of the access paths sharing it are live at $\pi$ . To solve the second problem, we need an availability analysis phase. It detects whether all links in the access path have been created along all execution paths reaching $\pi$. The results of these analysis are used to filter out those access paths whose nullification may be unsafe. We do not address the descriptions of these analyses in this paper.

## 6 Related Work

In this paper, we have described a static analysis for inferring dead references in first order functional programs. We employ a context free grammar based abstraction for the heap. This is in the spirit of the work by Jones and Muchnick [13] for functional programs. The existing literature related to improving memory efficiency of programs can be categorized as follows:

*Compile time reuse.* The method by Barth [2] detects memory cells with zero reference count and reallocates them for further use in the program. Jones and Le Metayer [15] describe a sharing analysis based garbage collection for reusing of cells. Their analysis incorporates liveness information: A cell is collected even when it is shared provided expressions sharing it do not need it for their evaluation.

*Explicit reclamation.* Shaham et. al. [25] use an automaton called *heap safety automaton* to model safety of inserting a free statement at a given program point. The analysis is based on shape analysis [20,21] and is very precise. The disadvantage of the analysis is that it is very inefficient and takes large time even for toy programs. *Free-Me* [7] combines a lightweight pointer analysis with liveness information that detects when short-lived objects die and insert statements to free such objects. The analysis is simpler and cheaper as the scope is limited. The analysis described by Inoue et. al. [12] detects the scope (function) out of which a cell becomes unreachable, and claims the cell using an explicit *reclaim* procedure whenever the execution goes out of that scope. Like our method, the result of their analysis is also represented using CFGs. The main difference between their work and ours is that we detect and nullify dead links at any point of the program, while they detect and collect objects that are unreachable at function boundaries. Cherem and Rugina [5] use a shape analysis framework [8] to analyze a single heap cell at a time for deallocation. However, multiple iterations of the analysis and the optimization steps are required, since freeing a cell might result in opportunities for more deallocations.

*Making dead objects unreachable.* The most popular approach to make dead objects unreachable is to identify live variables in the program to reduce the root set to only the live reference variables [1]. The major drawback of this approach is that all heap objects reachable from the live root variables are considered live, even if some of them may not be used by the program. *Escape analysis* [3,4,6] based approaches discover objects escaping a procedure (an escaping object being an object whose lifetimes outlives the procedure that created it). All non-escaping objects are allocated on stack, thus becoming unreachable whenever the creating procedure exits. In *Region* based garbage collection [9], a static analysis called *region inference* [26] is used to identify *regions* that are storage for objects. Normal memory blocks can be allocated at any point in time; they are always allocated in a particular region and are deallocated at the end of that region's lifetime. Approaches based on escape analysis and region inference detect garbage only at the boundaries of certain predefined areas of the program. In our previous work [17], we have used bounded abstractions of access paths called *access graphs* to describe the liveness of memory links in imperative programs and have used this information to nullify dead links.

A related work due to Heine and Lam [10] attempts to find potential memory leaks in C/C++ programs by detecting the earliest point in a program when an object becomes unreachable.

## 7 Conclusions

In this paper we presented a technique to compute liveness of heap data in functional programs. This information could be used to nullify links in heap memory to improve garbage collection. We have abstracted the liveness information in the form of a CFG, which is then converted to NFAs. This conversion implies some imprecision. We present a novel way to simplify the NFAs so they directly describe paths in the heap. Unlike the method described by Inoue et. al. [12], our simplification does not cause any imprecision.

In future, we intend to take this method to its logical conclusion by addressing the issue of nullification. This would require us to perform alias analysis which we feel can be done in a similar fashion. We also feel that with minor modification our method can be used for dead code elimination and intend to extend our analysis to higher order languages.

## References


[1] Agesen, O., D. Detlefs and J. E. Moss, *Garbage collection and local variable type-precision and liveness in Java virtual machines*, in: *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation* (1998), pp. 269–279.

[2] Barth, J. M., *Shifting garbage collection overhead to compile time*, Commun. ACM **20** (1977), pp. 513–518.

[3] Blanchet, B., *Escape analysis for object-oriented languages: application to Java*, in: *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (1999), pp. 20–34.

[4] Blanchet, B., *Escape analysis for Java$^{TM}$: Theory and practice*, ACM Transactions on Programming Languages and Systems **25** (2003), pp. 713–775.

[5] Cherem, S. and R. Rugina, *Compile-time deallocation of individual objects*, in: *ISMM '06: Proceedings of the 2006 international symposium on Memory management* (2006), pp. 138–149.

[6] Choi, J.-D., M. Gupta, M. Serrano, V. C. Sreedhar and S. Midkiff, *Escape analysis for Java*, in: *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (1999), pp. 1–19.

[7] Guyer, S. Z., K. S. McKinley and D. Frampton, *Free-me: a static analysis for automatic individual object reclamation*, in: *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation* (2006), pp. 364–375.

[8] Hackett, B. and R. Rugina, *Region-based shape analysis with tracked locations*, in: *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (2005), pp. 310–323.

[9] Hallenberg, N., M. Elsman and M. Tofte, *Combining region inference and garbage collection*, in: *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation* (2002), pp. 141–152.

[10] Heine, D. L. and M. S. Lam, *A practical flow-sensitive and context-sensitive c and c++ memory leak detector*, in: *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation* (2003), pp. 168–181.

[11] Hopcroft, J. E. and J. D. Ullman, "Introduction To Automata Theory, Languages, And Computation," Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[12] Inoue, K., H. Seki and H. Yagi, *Analysis of functional programs to detect run-time garbage cells*, ACM Trans. Program. Lang. Syst. **10** (1988), pp. 555–578.

[13] Jones, N. D. and S. S. Muchnick, *Flow analysis and optimization of lisp-like structures*, in: *POPL '79: Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages* (1979), pp. 244–256.

[14] Jones, R. and R. Lins, "Garbage collection: algorithms for automatic dynamic memory management," John Wiley & Sons, Inc., New York, NY, USA, 1996.

[15] Jones, S. B. and D. L. Metayer, *Compile-time garbage collection by sharing analysis*, in: *FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture* (1989), pp. 54–74.

[16] Karkare, A., A. Sanyal and U. Khedker, *Effectiveness of garbage collection in MIT/GNU scheme*, http://arxiv.org/abs/cs/0611093 (2006).

[17] Khedker, U., A. Sanyal and A. Karkare, *Heap reference analysis using access graphs*, Submitted to *ACM Transactions on Programming Languages and Systems*, copy available at http://arxiv.org/abs/cs.PL/0608104 (2006).

[18] Mohri, M. and M.-J. Nederhof, *Regular approximation of context-free grammars through transformation*, in: J.-C. Junqua and G. van Noord, editors, *Robustness in Language and Speech Technology*, Kluwer Academic Publishers, Dordrecht, 2000 pp. 251–261.

[19] Röjemo, N. and C. Runciman, *Lag, drag, void and use—heap profiling and space-efficient compilation revisited*, in: *ICFP '96: Proceedings of the first ACM SIGPLAN international conference on Functional programming* (1996), pp. 34–41.

[20] Sagiv, M., T. Reps and R. Wilhelm, *Parametric shape analysis via 3-valued logic*, in: *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (1999), pp. 105–118.

[21] Sagiv, M., T. Reps and R. Wilhelm, *Parametric shape analysis via 3-valued logic*, ACM Transactions on Programming Languages and Systems **24** (2002), pp. 217–298.

[22] Shaham, R., E. K. Kolodner and M. Sagiv, *On the effectiveness of gc in java*, in: *ISMM '00: Proceedings of the 2nd international symposium on Memory management* (2000), pp. 12–17.

[23] Shaham, R., E. K. Kolodner and M. Sagiv, *Heap profiling for space-efficient java*, in: *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation* (2001), pp. 104–113.

[24] Shaham, R., E. K. Kolodner and M. Sagiv, *Estimating the impact of heap liveness information on space consumption in Java*, in: *ISMM '02: Proceedings of the 3rd international symposium on Memory management* (2002), pp. 64–75.

[25] Shaham, R., E. Yahav, E. K. Kolodner and M. Sagiv, *Establishing local temporal heap safety properties with applications to compile-time memory management*, Sci. Comput. Program. **58** (2005), pp. 264–289.

[26] Tofte, M. and L. Birkedal, *A region inference algorithm*, ACM Transactions on Programming Languages and Systems **20** (1998), pp. 724–767.

| Program Point ($\pi$) | Live Access Patterns for $e$ at $\pi$ ($\sigma$) | Liveness Environment after $e$ ($\mathcal{L}$) | Liveness Environment at $\pi$ ($\mathcal{XE}(e,\sigma,\mathcal{L})$) |
|---|---|---|---|
| $\pi_1$ | $\sigma$ | $\emptyset$ | $\left\{\begin{array}{c}\mathsf{list1}.(\{\epsilon\} \cup \{\mathbf{0\bar{0}}\} \cdot \sigma \cup \{\mathbf{1}\} \cdot \mathcal{XF}^1_{\mathsf{app}}(\{\mathbf{\bar{1}}\} \cdot \sigma)),\\ \mathsf{list2}.(\sigma \cup \mathcal{XF}^2_{\mathsf{app}}(\{\mathbf{\bar{1}}\} \cdot \sigma))\end{array}\right\}$ |
| $\pi_2$ | $\{\epsilon\}$ | $\left\{\begin{array}{c}\mathsf{list1}.(\{\epsilon\} \cup \{\mathbf{0\bar{0}}\} \cdot \sigma \cup \{\mathbf{1}\} \cdot \mathcal{XF}^1_{\mathsf{app}}(\{\mathbf{\bar{1}}\} \cdot \sigma)),\\ \mathsf{list2}.(\sigma \cup \mathcal{XF}^2_{\mathsf{app}}(\{\mathbf{\bar{1}}\} \cdot \sigma))\end{array}\right\}$ | $\left\{\begin{array}{c}\mathsf{list1}.(\{\epsilon\} \cup \{\mathbf{0\bar{0}}\} \cdot \sigma \cup \{\mathbf{1}\} \cdot \mathcal{XF}^1_{\mathsf{app}}(\{\mathbf{\bar{1}}\} \cdot \sigma)),\\ \mathsf{list2}.(\sigma \cup \mathcal{XF}^2_{\mathsf{app}}(\{\mathbf{\bar{1}}\} \cdot \sigma))\end{array}\right\}$ |
| $\pi_3$ | $\{\epsilon\}$ | $\left\{\begin{array}{c}\mathsf{list1}.(\{\epsilon\} \cup \{\mathbf{0\bar{0}}\} \cdot \sigma \cup \{\mathbf{1}\} \cdot \mathcal{XF}^1_{\mathsf{app}}(\{\mathbf{\bar{1}}\} \cdot \sigma)),\\ \mathsf{list2}.(\sigma \cup \mathcal{XF}^2_{\mathsf{app}}(\{\mathbf{\bar{1}}\} \cdot \sigma))\end{array}\right\}$ | $\left\{\begin{array}{c}\mathsf{list1}.(\{\epsilon\} \cup \{\mathbf{0\bar{0}}\} \cdot \sigma \cup \{\mathbf{1}\} \cdot \mathcal{XF}^1_{\mathsf{app}}(\{\mathbf{\bar{1}}\} \cdot \sigma)),\\ \mathsf{list2}.(\sigma \cup \mathcal{XF}^2_{\mathsf{app}}(\{\mathbf{\bar{1}}\} \cdot \sigma))\end{array}\right\}$ |
| $\pi_4$ | $\sigma$ | $\left\{\begin{array}{c}\mathsf{list1}.(\{\epsilon\} \cup \{\mathbf{0\bar{0}}\} \cdot \sigma \cup \{\mathbf{1}\} \cdot \mathcal{XF}^1_{\mathsf{app}}(\{\mathbf{\bar{1}}\} \cdot \sigma)),\\ \mathsf{list2}.\mathcal{XF}^2_{\mathsf{app}}(\{\mathbf{\bar{1}}\} \cdot \sigma)\end{array}\right\}$ | $\left\{\begin{array}{c}\mathsf{list1}.(\{\epsilon\} \cup \{\mathbf{0\bar{0}}\} \cdot \sigma \cup \{\mathbf{1}\} \cdot \mathcal{XF}^1_{\mathsf{app}}(\{\mathbf{\bar{1}}\} \cdot \sigma)),\\ \mathsf{list2}.(\sigma \cup \mathcal{XF}^2_{\mathsf{app}}(\{\mathbf{\bar{1}}\} \cdot \sigma))\end{array}\right\}$ |
| $\pi_5$ | $\sigma$ | $\emptyset$ | $\left\{\begin{array}{c}\mathsf{list1}.(\{\epsilon\} \cup \{\mathbf{0\bar{0}}\} \cdot \sigma \cup \{\mathbf{1}\} \cdot \mathcal{XF}^1_{\mathsf{app}}(\{\mathbf{\bar{1}}\} \cdot \sigma)),\\ \mathsf{list2}.\mathcal{XF}^2_{\mathsf{app}}(\{\mathbf{\bar{1}}\} \cdot \sigma)\end{array}\right\}$ |
| $\pi_6$ | $\{\mathbf{\bar{0}}\} \cdot \sigma$ | $\left\{\begin{array}{c}\mathsf{list1}.\{\mathbf{1}\} \cdot \mathcal{XF}^1_{\mathsf{app}}(\{\mathbf{\bar{1}}\} \cdot \sigma),\\ \mathsf{list2}.\mathcal{XF}^2_{\mathsf{app}}(\{\mathbf{\bar{1}}\} \cdot \sigma)\end{array}\right\}$ | $\left\{\begin{array}{c}\mathsf{list1}.(\{\epsilon\} \cup \{\mathbf{0\bar{0}}\} \cdot \sigma \cup \{\mathbf{1}\} \cdot \mathcal{XF}^1_{\mathsf{app}}(\{\mathbf{\bar{1}}\} \cdot \sigma)),\\ \mathsf{list2}.\mathcal{XF}^2_{\mathsf{app}}(\{\mathbf{\bar{1}}\} \cdot \sigma)\end{array}\right\}$ |
| $\pi_7$ | $\{\epsilon\} \cup \{\mathbf{0\bar{0}}\} \cdot \sigma$ | $\left\{\begin{array}{c}\mathsf{list1}.\{\mathbf{1}\} \cdot \mathcal{XF}^1_{\mathsf{app}}(\{\mathbf{\bar{1}}\} \cdot \sigma),\\ \mathsf{list2}.\mathcal{XF}^2_{\mathsf{app}}(\{\mathbf{\bar{1}}\} \cdot \sigma)\end{array}\right\}$ | $\left\{\begin{array}{c}\mathsf{list1}.(\{\epsilon\} \cup \{\mathbf{0\bar{0}}\} \cdot \sigma \cup \{\mathbf{1}\} \cdot \mathcal{XF}^1_{\mathsf{app}}(\{\mathbf{\bar{1}}\} \cdot \sigma)),\\ \mathsf{list2}.\mathcal{XF}^2_{\mathsf{app}}(\{\mathbf{\bar{1}}\} \cdot \sigma)\end{array}\right\}$ |
| $\pi_8$ | $\{\mathbf{\bar{1}}\} \cdot \sigma$ | $\emptyset$ | $\left\{\begin{array}{c}\mathsf{list1}.\{\mathbf{1}\} \cdot \mathcal{XF}^1_{\mathsf{app}}(\{\mathbf{\bar{1}}\} \cdot \sigma),\\ \mathsf{list2}.\mathcal{XF}^2_{\mathsf{app}}(\{\mathbf{\bar{1}}\} \cdot \sigma)\end{array}\right\}$ |
| $\pi_9$ | $\mathcal{XF}^1_{\mathsf{app}}(\{\mathbf{\bar{1}}\} \cdot \sigma)$ | $\{\mathsf{list2}.\mathcal{XF}^2_{\mathsf{app}}(\{\mathbf{\bar{1}}\} \cdot \sigma)\}$ | $\left\{\begin{array}{c}\mathsf{list1}.\{\mathbf{1}\} \cdot \mathcal{XF}^1_{\mathsf{app}}(\{\mathbf{\bar{1}}\} \cdot \sigma),\\ \mathsf{list2}.\mathcal{XF}^2_{\mathsf{app}}(\{\mathbf{\bar{1}}\} \cdot \sigma)\end{array}\right\}$ |
| $\pi_{10}$ | $\{\mathbf{1}\} \cdot \mathcal{XF}^2_{\mathsf{app}}(\{\mathbf{\bar{1}}\} \cdot \sigma)$ | $\{\mathsf{list2}.\mathcal{XF}^2_{\mathsf{app}}(\{\mathbf{\bar{1}}\} \cdot \sigma)\}$ | $\left\{\begin{array}{c}\mathsf{list1}.\{\mathbf{1}\} \cdot \mathcal{XF}^1_{\mathsf{app}}(\{\mathbf{\bar{1}}\} \cdot \sigma),\\ \mathsf{list2}.\mathcal{XF}^2_{\mathsf{app}}(\{\mathbf{\bar{1}}\} \cdot \sigma)\end{array}\right\}$ |
| $\pi_{11}$ | $\mathcal{XF}^2_{\mathsf{app}}(\{\mathbf{\bar{1}}\} \cdot \sigma)$ | $\emptyset$ | $\{\mathsf{list2}.\mathcal{XF}^2_{\mathsf{app}}(\{\mathbf{\bar{1}}\} \cdot \sigma)\}$ |

Computation of liveness environment for app

# Separation Analysis for Deductive Verification [1]

Thierry Hubert [2,3,5] Claude Marché [4,5]

*INRIA Futurs - ProVal*
*Parc Orsay Université - ZAC des Vignes*
*3, rue Jacques Monod - Bâtiment N*
*F-91893 ORSAY Cedex*

**Abstract**

The component-as-array model is a widely used technique for modeling heap memory in order to perform deductive verification of pointer programs. We propose an separation analysis which can be integrated in the core of this model. This allows to greatly simplify the verification conditions generated by a weakest precondition calculus, and thus greatly helps proving such pointer programs. We illustrate the improvements both in term of scaling up for codes of large size, and in term of simplification of the reasoning for establishing advanced behaviors.

*Keywords:* Deductive Verification, Separation Analysis, Regions, Polymorphism

## 1 Introduction

To perform verification of pointer programs, it is widely known that detection of pointer aliasing is essential. A separation analysis is a technique to automatically detect that two given pointers are not alias to each other. Various separation analyzes have been proposed, in the context of advanced static analysis of programs and abstract interpretation.

Deductive verification is the class of verification techniques that are based on logical semantics of programs, starting from the landmark work of Floyd and Hoare [1,2], and the concepts of logical assertions such as pre- and post-conditions, loop invariants, etc. Compared to static analysis techniques, deductive verification is potentially much more expressive, and is able to establish advanced behaviors of programs, the main drawback being that logic assertions must be given by the programmer.

The most well-known technique for analysing separation in the context of deductive verification is Separation Logic proposed by Reynolds [3]. Yet, as far as

[1] This research is partly supported by “CIFRE” contract 774/2004 with Dassault Aviation company, and ANR RNTL grant “CAT”

[2] Dassault Aviation, Suresnes, France

[3] CNRS, Laboratoire de Recherche en Informatique, UMR8623, Orsay, F-91405

[4] INRIA Futurs, Orsay, F-91893

[5] Univ Paris-Sud, Orsay, F-91405

we know, no tool implementing Separation Logic has demonstrated a disruptive progress on reasoning on concrete case studies such as industrial embedded code.

For deductive verification, the technique that has shown itself the most effective in practice is the Weakest Precondition (WP) calculus of Dijkstra [4]. It is the base of effective tools such as ESC/Java [5], several tools for Java programs annotated using the Java Modeling Language [6], Spec# [7] for the C# programming language, and a tool of our own called Caduceus [8] for C programs.

Dealing with pointer programs in the context of a WP calculus is usually done by providing an appropriate axiomatic modeling of the memory heap. The component-as-array model, coming from an old idea by Burstall [9], has been emphasized by Bornat [10]. Variants of this modeling are used by tools mentioned above. Unfortunately Separation Logic is not easily compatible with those techniques based on WP and the component-as-array modeling.

The goal of this paper is to propose a new separation analysis, based on the idea of separation logic, but directly suitable for the component-as-array modeling. The idea is quite natural: memory regions that are guaranteed to be separated can be modelled by distinct components of the heap memory model. The method we present here is general and applicable to many programming languages: we present it for C but it is clearly possible to apply it to Java, C#, etc. In Section 2, we define the core language we consider, and recall the principles of the component-as-array modeling.

In Section 3, we present our new separation analysis, and define the refined component-as-array model it leads to. The main technique is based on a polymorphic type system à la Milner [11]. Indeed, in our implementation inside the Caduceus tool, this type system is explicitely used, because internally Caduceus produces intermediate representation of the program into the Why language [12], which is itself a polymorphically typed language.

In Section 4, we show applications of the technique, and experimental results obtained with our implementation in the Caduceus tool for C. We compare to related work in Section 5 and conclude in Section 6.

## 2 Preliminaries

### *2.1 Core language*

Our analysis is described on a core language. Data types of this language are only integers and structures.

Core expressions are made of constants, variables, standard operators, function calls, field accesses, pointer arithmetic. Other C constructs can be translated to these.

$$
\begin{aligned}
e ::= {} & c && \text{(constants)} \\
| {} & v && \text{(variables)} \\
| {} & e \ op \ e && \text{(integer operators +,-,*,/,\%, \&\&, etc.)} \\
| {} & e \rightarrow f && \text{(field access)} \\
| {} & id(e, \ldots, e) && \text{(function call)}
\end{aligned}
$$

$\mid e \oplus e$ (addition pointer+integer giving a pointer)
$\mid e \ominus e$ (subtraction pointer-pointer giving an integer)

Statements are

$s ::= v = e;$ (variable assignment)
$\mid e \rightarrow f = e;$ (field assignment)
$\mid$ **return** $e;$ (function return)
$\mid$ **if**$(e)$ $s$ **else** $s$ (conditional branching)
$\mid$ **while**$(e)$ $s$ (while loop)

We indeed have such a language in the implementation of Caduceus, with some others constructs like switch, break, etc. Only constructs given above are important for the rest of the paper. We also support allocation and deallocation, but we ignore them first simplicity. We also ignore procedure calls, which can be treated similarly as function calls.

### *2.2 Normalization of C source*

We present briefly how we transform general C code into our core language. The two main points are to remove the address operator &, and to reduce the star operator $*p$, array accesses $t[i]$, and dot fields accesses $e.f$ to arrow field access $e \rightarrow f$. The address operator is removed by an initial analysis which for each variable $x$ present as argument of &, transformes it into a pointer variable (or more precisely an array of size 1) to the type of $x$ in the original C code ; and then each occurrence of $\&x$ becomes $x$ and each occurrence of $x$ becomes $*x$. The same is done for structure fields: if the address of a structure field $f$ is taken somewhere as in $\&e.f$, then the type of $f$ becomes a pointer (more precisely an array of size 1) to the type of $f$ in the original C code. Each occurrence of $\&e.f$ becomes $e.f$ and $e.f$ becomes $e \rightarrow f$ (because it is equivalent to $*(e.f)$). Star operators, array accesses, and remaining dot field accesses are then reduced by the following normalization rules:

$$\overline{e.f} = \overline{e} \rightarrow f$$
$$\overline{*e} = \overline{e} \rightarrow F(e)$$
$$\overline{e_1[e_2]} = (\overline{e_1} \oplus \overline{e_2}) \rightarrow F(e_1)$$

where $F(e)$ is a field name generated from the type of $e$: for the `int` type we use `intM` (for "int memory"), for type `int*` we use `intPM` ("int pointer memory"), etc.

Figure 1 shows an example of normalization of C code.

### *2.3 Component-as-array modeling*

The key idea proposed by Burstall [9] is to have one 'array' variable for each structure field, which is indeed an applicative map which can be accessed or modified only via two side-effect free functions *select* and *store*, which satisfy the so-called *theory of arrays* :

(1) $select(store(a, i, v), i) = v$
(2) $select(store(a, i, v), j) = select(a, j)$ if $i \neq j$

This modeling syntactically encodes the fact that two structure fields cannot be aliased. The important consequence is that whenever one field is updated, only the corresponding array variable is modified and we have for free that any other field is left untouched.

Filliâtre and Marché proposed a variant of this technique to deal with C pointer arithmetic [8]. The C memory heap is also represented by a finite set of array variables, indexed by *pointers* viewed as pairs of an adress to an allocated block and an offset into this block. Thus 'array' variables are indeed 2-dimensional. For our core language, since we have the shift operation on pointers as a primitive, we indeed do not need anymore to make explicit this 2-dimensional representation.

In the same paper, the modeling of the heap is described using multi-sorted polymorphic first-order logic. This logic is indeed the logic of the Why tool, which is a verification tool based on a WP calculus, for a WHILE language with only *non-aliased* global variables [12].

So, with our core language, modeling of memory is done by introducing two logic sorts: `pointer` and $\alpha$ `memory`, and operations

$$\begin{aligned} shift &: \texttt{pointer}, \texttt{integer} \rightarrow \texttt{pointer} \\ select &: \alpha\ \texttt{memory}, \texttt{pointer} \rightarrow \alpha \\ store &: \alpha\ \texttt{memory}, \texttt{pointer}, \alpha \rightarrow \alpha\ \texttt{memory} \end{aligned}$$

satisfying the theory of arrays given above and

$$\begin{aligned} shift(p,0) &= p \\ shift(shift(p,i),j) &= shift(p,i+j) \end{aligned}$$

Then an interpretation of our core language into a WHILE language is given by transformation rules:

$$\begin{aligned} [e \rightarrow f] &= select(f,[e]) \\ [e_1 \oplus e_2] &= shift([e_1],[e_2]) \\ [v = e] &= v := [e] \\ [e_1 \rightarrow f = e_2] &= f := store(f,[e_1],[e_2]) \end{aligned}$$

Statements are interpreted into WHILE constructs, and memory accesses $e \rightarrow f$ are in fact guarded with an assertion to check validity of pointer dereferencing. This

Original code:

```
int x;
int t[2];
struct S { int y;} s;

void f() {
  int *z = &x;
  t[1] = *z;
  s.y = t[1];
}
```

Normalized code:

```
int x[1];
int t[2];
struct S { int y; };
struct S s[1];

void f() {
  int *z = x;
  (t ⊕ 1)->intM = z->intM;
  s->y = (t ⊕ 1)->intM;
}
```

Fig. 1. Example of C code normalization

```
struct S { int i; };

/*@ requires \valid(x) && \valid(y)
  @ assigns x->i, y->i
  @ ensures x->i == 1 && y->i == 2
  @*/
void f(struct S *x, struct S *y) {
  x->i = 1; y->i = 2;
}

struct S t1[1], t2[1];

//@ ensures t1[0].i == 1 && t2[0].i == 2
void g() { f(&t1[0],&t2[0]); }
```

Fig. 2. Simple case of separation analysis

aspect is not useful for the remaining, so we refer to [8,13] for details.

## 3 Separation Analysis

### *3.1 Modeling with regions*

We want to integrate a separation analysis into the modeling presented above. Let's illustrate that on simple examples. Consider the C code of Figure 2. We use here the syntax of the Caduceus specification language, which is very similar to JML: the annotations are given in special comments `/*@ .. */`, **requires** introduces a pre-condition, **ensures** a post-condition, and **assigns** is a clause to specify which memory location are modified [8,13]. The annotation `\valid(x)` means that $x$ points to a safely allocated memory location.

Post-condition of function `f` cannot be established, because it is indeed wrong if pointers `x` and `y` appear to be alias, that is if they are equal. For the call to `f` in function `g` they are different, but since we use a modular reasoning (function by function, as for any technique based on WP), the whole code cannot be proven correct. A possible solution could be to add to the pre-condition of `f` the additional hypothesis `x != y`, but our goal in this paper is to avoid this extra condition. The WHILE interpretation of the C code of `f` is

```
i := store(i,x,1) ; i := store(i,y,2) ;
```

so establishing the post-condition $select(i, x) == 1$ amounts to prove

$$select(store(store(i, x, 1), y, 2), x) = 1$$

which is a consequence of axioms 1 and 2 if $x \neq y$. Our goal is to interpret the code of $f$ differently using two distinct variables for representing the fields of $x$ and $y$:

$$i_x := store(i_x, x, 1); i_y := store(i_y, y, 2);$$

These are two distinct *regions* for field $i$. With that interpretation, post-conditions $select(i_x, x) = 1$ and $select(i_y, y) = 2$ are consequences of axiom 1 without the need of $x \neq y$.

Consider additionally the code of Figure 3. Function `f` is now called twice, in the first call $x$ points to array $s.t1$ and $y$ points to array $s.t2$, and it is reversed in the second call. To allow the use of different regions, we need to make these regions *parameters* to f, and we call them *parametric regions*. The complete interpretations

```
struct T {
  struct S t1[2];
  struct S t2[2];
};

/*@ ensures s.t1[0].i == 1 && s.t2[0].i == 2 &&
  @         s.t1[1].i == 2 && s.t2[1].i == 1
  @*/
void h(struct T s) {
  f(&s.t1[0],&s.t2[0]);
  f(&s.t2[1],&s.t1[1]);
}
```

Fig. 3. Case of parametric regions

of $f$ and $h$ are then

```
  void f(i_x,i_y,x,y) {
    i_x := store(i_x,x,1) ; i_y := store(i_y,y,2) ;
  }
```

and

```
void h(t1,t2,i_t1,i_t2,s) {
     f(i_t1,i_t2,select(t1,s),select(t2,s));
     f(i_t2,i_t1,shift(select(t2,s),1),shift(select(t1,s),1));
  }
```

and their post-conditions can be established by simple first-order reasoning. Notice that this holds because our WHILE back-end language assumes non-aliased variables, so that $i_x$ and $i_y$ are assumed distinct. Indeed, the Why tool we use to interpret this intermediaire statically checks this non-aliasing: an attempt to call $f$ with the same value for $i_x$ and $i_y$ would be rejected, as an ill-typed program.

So our goal is to integrate a notion of separation into the modeling of C code, by attaching regions to pointers and memory variables. The interpretation of a memory access $e \rightarrow f$ is now $\mathit{select}(f_r, e)$ where $r$ is the region of $e$. We see now how we compute those regions.

### *3.2 Regions as types*

We see regions as a rich type system for pointers. For simplicity, we only consider pointers and the `int` base type. The types of expressions are then given by the grammars

$$
\begin{array}{llll}
\text{(types)} & \tau ::= \texttt{int} & & \\
 & \quad \mid r\ \texttt{pointer} & \text{(pointer to region } r) & \\
 & \quad \mid (\tau, r)\ \texttt{memory} & \text{(memory of values of type } \tau \text{ in region } r) & \\
\text{(regions)} & r ::= \rho & \text{(region variable)} & \\
 & \quad \mid R & \text{(region constant)} &
\end{array}
$$

Region variables are needed for the parametric regions passed to functions: regions as function parameters have a *polymorphic* type. Our type system is then just a particular case of a polymorphic type system *à la* Milner [11].

If we consider the function $f$ of example above, its profile is

$$f(i_x : (int, \rho_1)\ \texttt{memory}, i_y : (int, \rho_2)\ \texttt{memory}, x : \rho_1\ \texttt{pointer}, y : \rho_2\ \texttt{pointer})$$

that is polymorphic in $\rho_1, \rho_2$: for each call to $f$ these regions can be instantiated differently.

### 3.2.1 *Region typing rules*

We now express separation by giving typing rules for expressions, using types with regions. The typing environment is made of two parts denoted $\Gamma$ and $\Delta$. $\Gamma$ is a classical typing environment which maps variable identifiers to types: we denote $x : t \in \Gamma$ whenever $\Gamma$ maps variable $x$ to the type $t$. $\Delta$ is a *region environment* which maps pairs $(r, f)$ to types, where $r$ is a region and $f$ is a field identifier. We denote that as $(r, f) : t \in \Delta$.

We are now able to give typing rules for expressions:

Integer constants:

$$\frac{}{\Gamma, \Delta \vdash n : \texttt{int}}$$

Type of a variable follows the environment $\Gamma$:

$$\frac{}{\Gamma, \Delta \vdash x : t} \text{ if } x : t \in \Gamma$$

Type of a field access follows the environment $\Delta$:

$$\frac{\Gamma, \Delta \vdash l : r\ \texttt{pointer}}{\Gamma, \Delta \vdash l \rightarrow f : t} \text{ if } (r, f) : t \in \Delta$$

Function calls:

$$\frac{\Gamma, \Delta \vdash e_1 : t_1 \quad \cdots \quad \Gamma, \Delta \vdash e_n : t_n}{\Gamma, \Delta \vdash id(e_1, \ldots, e_n) : t}$$

if $id : (\tau_1, \cdots, \tau_n) \rightarrow \tau \in \Gamma$ and there is a region substitution $\sigma$ such that $t = \tau\sigma$ and for each $i$, $t_i = \tau_i\ \sigma$ (polymorphic typing).

Pointer shift keeps the same region:

$$\frac{\Gamma, \Delta \vdash e_1 : r\ \texttt{pointer} \quad \Gamma, \Delta \vdash e_2 : \texttt{int}}{\Gamma, \Delta \vdash e_1 \oplus e_2 : r\ \texttt{pointer}}$$

Difference and comparison of pointers must be done only with pointers in the same region:

$$\frac{\Gamma, \Delta \vdash e_1 : r\ \texttt{pointer} \quad \Gamma, \Delta \vdash e_2 : r\ \texttt{pointer}}{\Gamma, \Delta \vdash e_1\ op\ e_2 : int}$$

where $op \in \{\ominus, ==, <=, =>, <, >, != \}$.

The typing rules for statements are then the following:

Variable assignment:

$$\frac{\Gamma, \Delta \vdash e : t}{\Gamma, \Delta \vdash v = e : t} \text{ if } x : t \in \Gamma$$

Field assignment:

$$\frac{\Gamma, \Delta \vdash e_1 : r\ \texttt{pointer} \quad \Gamma, \Delta \vdash e_2 : t}{\Gamma, \Delta \vdash e_1 \rightarrow f = e_2 : t} \text{ if } (r, f) : t \in \Delta$$

Typing of other statements is done in a natural way.

For the typing of local or global declarations, we assume for the moment given an oracle which gives the regions involved in the construction of $\Gamma$. For declarations of structures, this means that the $\Delta$ environment is also given. In other words, typing of functions is made in a given $\Delta$.

The first result we give is a soundness property of the typing rules: indeed this soundness is relative in the sense that it shows that the interpretations of programs are the same with or without the separation of memory variables.

**Theorem 3.1 (Relative soundness)** *If a program is well typed in a given environment $\Gamma, \Delta$, then its logical interpretation with region memory variables has the same semantics as its interpretation with the classical component-as-array model.*

Proof sketch: we provide a bisimulation of execution steps of interpreted programs, with or without separation. A state of the program with regions can be seen as a partition of the state of the program without regions. Each operation on the program with region can be simulated on the state of the program without region and vice-versa: this works because all operations respect the partition, because the program is well-typed in term of regions.

### *3.3 Inference of regions*

The remaining step is now to provide an inference system to construct an environment $\Gamma, \Delta$ which makes a given program well-typed, if possible.

Since our type system is a particular case of a polymorphic type system à la Milner, we can derive an inference method from known type inference algorithms such as the W algorithm [14]. The only specific feature is the handling of the $\Delta$ part of the environment.

In a first step, we assign to each global pointer variable a fresh region constant. Parameters of functions, and local variables, which are pointers, are given a fresh region variable. This provides an initial $\Gamma$. We build at the same time an initial $\Delta$ which makes everything separated *a priori*.

In a second step, functions are analyzed, in the order given by the call graph, to determine their polymorphic type. For each function, the code is traversed, and the typing rules given above lead to equality constraints between regions, that is we perform unification of regions. Each time a function is analyzed, we determine which of the region variables remain not instantiated, and we make the function type polymorphic by quantifying over them.

Unification of regions is standard, expect for the handling of $\Delta$: each time two regions $r_1$ and $r_2$ are made identical, we need to perform a merge operation on $\Delta$: for each field $f$ such that $\Delta$ maps $(r_1, f)$ to $t_1$ and $(r_2, f)$ to $t_2$, we need to merge the mappings, and consequently unify the type $t_1$ and $t_2$ (which may recursively lead to unification of other regions). During this unification phase, $\Gamma$ is also modified by side-effect.

At the end of this process, we end up with a $\Gamma$ and a $\Delta$ in which the program is well-typed.

This type inference process indeed computes the separation into the largest possible number of regions, allowed by the typing rules given.

# 4 Applications

Our separation analysis is implemented in the Caduceus tool, as a user option. Selection of this option asks to perform the inference of regions, and then the generation of the model and the Why interpretation of C code is modified accordingly. We show here a few experiments and applications.

## *4.1 Caduceus benchmarks*

Caduceus benchmarks is a set of small C programs that are used as a non-regression test. These examples are small, and most of them are motivated by other concerns than separation.

Without the separation analysis, for the whole set of examples there is a total number of 1324 verification conditions generated. These are passed to the automatic theorem prover Simplify [15], and 1287 of them are discharged, that is 97.2%.

With the separation analysis turned on, there are 1349 verification conditions generated. At first, it seems that the number of them should be the same, because separation analysis leads to Why interpretations of programs which have exactly the same structure. Indeed, the number of them is different because of two reasons:

- First, trivial verification conditions (mainly propositional tautologies) are indeed automatically discharged silently, and in some cases, with the separation analysis some verification conditions are tautologies whereas without separation they are not. So this may make the number of VCs smaller.
- Second, one has to notice that the interpretation of **assigns** clauses produces as many propositions as the number of memory variables involved, which is larger when separation is turned on. So this may make the number of VCs larger, but each of them is simpler.

With separation analysis on, the automatic theorem prover Simplify discharges 1327 VCs, that is 98.3%, a slightly better score.

This means that separation analysis helps sometimes for those small examples, but more importantly, this means that this does not bring overhead on examples where separation is not the concern. Remark also that the separation analysis itself is quick (we believe it is linear in the size of the code), so probably separation analysis could be turned on by default in the future.

## *4.2 Regions and logical annotations*

This example is inspired from a piece of Java code by P. Müller [16]. It computes the set of positive elements of an array, and puts them in a new array.

```
int *m(int t[], int length) {
  int count = 0; int i; int *u;

  for (i=0; i < length; i++) if (t[i] > 0) count++;

  u = (int*)calloc(count,sizeof(int));
  count = 0;
```

```
  for (i=0; i < length; i++) if (t[i] > 0) u[count++] = t[i];
  return u;
}
```

To verify that the assignment of `u[count]` is inside the array bounds is tricky: it involves a "semantic" reasoning, noticing that the second loop counts exactly the same number of elements as the first, so the index `count` must be smaller than the value of `count` used for allocating the array `u`. To make this reasoning explicit, it is natural to annotate the loops with an invariant, the same one for both loops:

```
  /*@ invariant
    @   count == \num_of(int j; 0 <= j && j < i ; t[j] > 0)
    @*/
  for (i=0 ; i < length; i++) ...
```

where `\num_of` is a JML-like construct [17] giving the number of elements satisfying the predicate given as argument. Indeed, the original example by Müller was precisely a challenge for static verification tools because none of them supports the `\num_of` construct. Anyway, it is possible on a given example to 'expand' the use of `\num_of`, and we did that for our C code: we introduce a *logic function* [8]:

```
//@ logic int num_of_pos(int i,int j,int a[]) reads t[..]
```

together with a few *axioms*:

```
/*@ axiom num_of_pos_empty :
  @   \forall int i, int j, int a[];
  @       i > j => num_of_pos(i,j,a) == 0
  @*/
/*@ axiom num_of_pos_true_case :
  @   \forall int i, int j, int k, int a[];
  @       i <= j && a[j] > 0 =>
  @         num_of_pos(i,j,a) == num_of_pos(i,j-1,a) + 1
  @*/
/*@ axiom num_of_pos_false_case :
  @   \forall int i, int j, int k, int a[];
  @       i <= j && ! (a[j] > 0) =>
  @         num_of_pos(i,j,a) == num_of_pos(i,j-1,a)
  @*/
```

(see http://www.lri.fr/~marche/MullerChallenge.pdf for the remaining annotations).

The key point now is that the verification of safety cannot be done, because in the second loop, we know that `count` is less than the number of positive elements in `t`, but there is a reasoning to perform to establish than this number of elements *did not change between the two loops*. With a single heap variable in the model for representing integer arrays, this is far from simple. Indeed, the logic function `num_of_pos` is axiomatized with some inductive scheme, and one should prove that it implies that `num_of_pos(i,j,a)` only depends on the values of `a[i..j]`, which is hard to prove.

On the other hand, with our modeling involving memory separation, it is statically detected that `t` and `u` are separated, and thus can be modeled with two separate heap variables `intM_t` and `intM_u`. Then, the logic function `num_of_pos` becomes *parametric* in the memory variable involved for the array argument `a`, and it becomes syntactically true that `num_of_pos(0,j-1,t)` is the same in both loop of our piece of C code. On that example, each verification condition is then discharged automatically by the Simplify prover.

### *4.3 An industrial case study*

In collaboration with Dassault Aviation company, we experimented Caduceus and its separation analysis on a real embedded code for avionics. The first experiment was made on a core of this code, which is approximately 3000 lines long. The characterics are that it contains a large number of data structures, and usually these structures contains nested arrays of other structures.

With this first experiment, we do not provide any user functional property, so that the only verifications made are for safety of pointer dereferencing and array accesses. Necessary function preconditions and loop invariants are indeed automatically generated by an ad-hoc tool using simple heuristics.

Without separation analysis, this code gives rise to 1151 VCs, 965 being discharged by Simplify, that is 83.8%. With separation, we get 1982 VCs, 1972 discharged by Simplify, that is 99.4 %. There is a total of 376 regions infered for global variables, and there are 242 polymorphic regions added as parameter to functions. The significantly higher number of VCs with separation analysis can be explained by the high number of regions, for the same reason mentioned above about **assigns** clauses. Notice that the 10 remaining VCs have been discharged by the interactive proof assistant Coq.

We believe this is a very positive experimental result, which shows that our separation analysis is a major improvement in practice. We are currently experimenting on the whole code (70000 lines long) with good results too. We are also trying to prove a complex behavioral property, involving logical annotations and ghost variables, for which we hope that the separation analysis will greatly simplify the reasoning, as on the previous example.

## 5 Related work

Talpin and Jouvelot proposed in 1994 [18] a calculus for analysing effects of programs based of a polymorphic type system. The principle is the same as ours, but their work is limited to reference variables: no deep sharing in data structures is possible.

In the context of static analysis, points-to analysis is a very advanced technique for computing information on pointers. This has been initiated by Andersen in 1994 [19] and extended further in 1996 by Steensgaard [20] and in 2000 by Das [21]. We used their idea of designing a type system for analysis separation, but it is clear that our analysis is much less precise than theirs. Our method is tailored to the generation of a refined component-as-array model for deductive verification.

The Cyclone system [22] proposes a new programming language analogous to C,

but in which the programmer can specify regions manually. They allow polymorphic regions in function call as us. But our setting applies to the real C language, and moreover regions are automatically inferred instead of being given by the user.

Compared to Separation Logic, our separation analysis is clearly less precise. It seems that Separation Logic is very powerful in some cases: it allows for example to specify that a linked list cannot be circular, or that a graph is a tree, and to reason with that. This is something that our analysis cannot do: as soon as one traverses a linked lists, only one region is inferred for the whole list. Our analysis is clearly more adapted to deal with programs with a high size of data, but we also have positive points for reasoning on small programs as shown by the example of Section 4.2. Combining all the power of Separation Logic and our separation analysis remains a future task.

## 6 Conclusion

We proposed a separation analysis that is potentially useful for any tool for deductive verification based on weakest precondition calculus and a component-as-array model. Our experimentations on C programs are very positive, as shown by applications given in Section 4.

There are some drawbacks that we plan to address in the future. First, the separation analysis must be done on the whole program, so it is not modular. This is not a major problem since the separation analysis is quick, and because after separation analysis is performed, the remaining of the verification task can still be done modularly, function by function. But in case we do not have the complete program available, such as if we what to verify libraries, this is a problem. We plan to add new constructs in the specification language to allow user specification of separation: for example, for a library function such as `memcpy`, one may want to specify that the source and target are separated.

Our separation analysis is tailored to its later use for the component-as-array model. In that model, a given array will be entirely in the same region. However, by advanced static analysis, it is possible to discover that an given array may be split into several regions, for example in the following code:

```
struct S  int x ;
struct S t[10];

void f(S *p, int n) { p->x = n; }

void main()  f(t+0,2); f(t+1,3);
```

we do not get for free that `t[0]->x == 2`. We are planning to incorporate more advanced static analysis techniques in our setting, in the context of the CAT project (http://www.rntl.org/projet/resume2005/cat.htm).

## References

[1] Floyd, R.W.: Assigning meanings to programs. In Schwartz, J.T., ed.: Mathematical Aspects of Computer Science. Volume 19 of Proceedings of Symposia in Applied Mathematics., Providence, Rhode

Island, American Mathematical Society (1967) 19–32

[2] Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM **12**(10) (1969) 576–580 and 583

[3] Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: 17h Annual IEEE Symposium on Logic in Computer Science, IEEE Comp. Soc. Press (2002)

[4] Dijkstra, E.W.: A discipline of programming. Series in Automatic Computation. Prentice Hall Int. (1976)

[5] Detlefs, D.L., Leino, K.R.M., Nelson, G., Saxe, J.B.: Extended static checking. Technical Report 159, Compaq Systems Research Center (1998) See also http://research.compaq.com/SRC/esc/.

[6] Burdy, L., Cheon, Y., Cok, D., Ernst, M., Kiniry, J., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML tools and applications. International Journal on Software Tools for Technology Transfer (2004)

[7] Leino, K.R.M.: Efficient weakest preconditions. Technical Report MSR-TR-2004-34, Microsoft Research (2004)

[8] Filliâtre, J.C., Marché, C.: Multi-prover verification of C programs. In Davies, J., Schulte, W., Barnett, M., eds.: Sixth International Conference on Formal Engineering Methods. Volume 3308 of Lecture Notes in Computer Science., Seattle, WA, USA, Springer-Verlag (2004) 15–29

[9] Burstall, R.: Some techniques for proving correctness of programs which alter data structures. Machine Intelligence **7** (1972) 23–50

[10] Bornat, R.: Proving pointer programs in Hoare logic. In: Mathematics of Program Construction. (2000) 102–126

[11] Milner, R.: A theory of type polymorphismn programming. Journal of Computer and System Sciences **17** (1978)

[12] Filliâtre, J.C.: Why: a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud (2003) http://www.lri.fr/~filliatr/ftp/publis/why-tool.ps.gz.

[13] Marché, C., Paulin-Mohring, C.: Reasoning about Java programs with aliasing and frame conditions. In Hurd, J., Melham, T., eds.: 18th International Conference on Theorem Proving in Higher Order Logics. Lecture Notes in Computer Science, Springer-Verlag (2005)

[14] Damas, L., Milner, R.: Principal type-schemes for functional programs. In: POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, New York, NY, USA, ACM Press (1982) 207–212

[15] Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. J. ACM **52**(3) (2005) 365–473

[16] Müller, P.: Specification and verification challenges. Exploratory Workshop: Challenges in Java Program Verification, Nijmegen, The Netherlands (2006) http://www.cs.ru.nl/~woj/esfws06/slides/Peter.pdf.

[17] Leavens, G.T., Leino, K.R.M., Poll, E., Ruby, C., Jacobs, B.: JML: notations and tools supporting detailed design in Java. In: OOPSLA 2000 Companion, Minneapolis, Minnesota. (2000) 105–106

[18] Talpin, J.P., Jouvelot, P.: Polymorphic type, region and effect inference. Journal of Functional Programming **2**(3) (1992) 245–271

[19] Andersen, L.O.: Program Analysis and Specialization for the C Programming Language. PhD thesis, University of Copenhagen (1994)

[20] Steensgaard, B.: Points-to analysis in almost linear time. In: Symposium on Principles of Programming Languages. (1996) 32–41

[21] Das, M.: Unification-based pointer analysis with directional assignments. In: PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation, New York, NY, USA, ACM Press (2000) 35–46

[22] Grossman, D., Morrisett, G., Jim, T., Hicks, M., Wang, Y., Cheney, J.: Region-based memory management in cyclone. In: PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation, New York, NY, USA, ACM Press (2002) 282–293

[23] Nanevski, A., Morrisett, G., Birkedal, L.: Polymorphism and separation in Hoare type theory. In Reppy, J.H., Lawall, J.L., eds.: 11th ACM SIGPLAN International Conference on Functional Programming, ICFP 2006, Portland, Oregon, USA, ACM (2006) 62–73

# Verifying Concurrent List–Manipulating Programs by LTL Model Checking

Joost–Pieter Katoen  Thomas Noll  Stefan Rieger

*RWTH Aachen University*
*Software Modeling and Verification Group*
*52056 Aachen, Germany*
{katoen,noll,rieger}@cs.rwth-aachen.de

**Abstract**

We present a novel approach to the verification of concurrent pointer–manipulating programs which operate on singly–linked lists. By abstracting from chains (i.e., non–interrupted sublists) in the heap, we obtain a finite–state representation of all possible executions of a given program. The combination of a simple pointer logic for expressing heap properties and of temporal operators then allows us to employ standard LTL model checking techniques. The usability of this approach is demonstrated by establishing correctness properties of a producer/consumer system and of a concurrent garbage collector.

*Keywords:* Software Model Checking, Abstraction, Heap Verification, Shape Analysis, LTL, Lists, Pointer Programs

## 1 Introduction

Techniques for the verification of elementary properties of concurrent pointer programs are indispensable. Programming with pointers is error–prone with potential pitfalls such as dereferencing null pointers and the creation of memory leaks. Pointer programming becomes even more vulnerable in a concurrent setting where data structures such as linked lists and trees are manipulated and inspected by several threads.

This paper presents a model–checking approach to the verification of concurrent programs that manipulate singly–linked lists. Existing approaches either make use of non–standard logics, advanced model–checking procedures or extended versions of Hoare logics with accompanying deduction techniques (see Sct. 6 about related work). In contrast, the approach advocated in this paper stays within the realm of traditional (linear–time) model checking. This facilitates the usage of standard (LTL) model checkers for validating temporal properties addressing absence of memory leaks, dereferencing of null pointers, dynamic creation of cells, and simple and position–dependent aliasing.

Our approach is illustrated by considering a simple concurrent programming language that besides the usual control structures offers primitives for pointer manipulation, cell creation and destruction, and (guarded) atomic regions that allow concurrency control constructs such as test–and–set primitives and monitors. An operational semantics is provided in terms of labeled transition systems in which states are equipped with a graph structure representing the current list configuration. List abstraction exploits a variant of summary nodes [45] that represent more than $M$ chained list cells where constant $M$ is directly obtained from the formula to be checked. Each configuration is shown to have a canonical representation (up to isomorphism). The abstract semantics of any concurrent program in our language is finite, obtained in a fully mechanized manner, and keeps the minimal "distance" between program variables and summary nodes invariant. Over–approximation oc-

curs in a very controlled manner; only assignments may yield nondeterminism as variables may get "too close" to summary nodes.

Properties are expressed in a first–order linear–time temporal logic (LTL) that is enriched with assertions on singly–linked lists such as reachability of cells, aliasing, and freshness of cells. Our logic is similar in spirit to NTL [19,20] and ETL [49]. Opposed to NTL, we avoid the use of temporal operators inside quantification. In this way, involved mechanisms to keep track of the identities of individual cells are not needed. As a result, standard LTL model checking algorithms can be employed. The differences with ETL are more of a technical nature. ETL has a three–valued interpretation, whereas our logical interpretation is a standard binary one. Moreover, ETL–formulas are translated in first–order logic with transitive closure for the evaluation on a trace, whereas in our case traces are generated by labeled transition systems and used in standard LTL model checking. The feasibility of our approach is shown by considering the verification of a simple concurrent garbage collection program. Furthermore a prototypical tool is currently under development for experimenting with real–life examples.

Please note that due to space constraints most of the proofs could not be included in this paper.

## 2 A List–Manipulating Programming Language

Given a universe $PV$ of program variables, we define the set of *list–manipulating programs* (LM–programs) to be given by the following grammar (where $v_i, v \in PV$):

$$
\begin{aligned}
\text{LMP} &::= \mathbf{var}\ v_1, ..., v_k(\text{Stmt}_1 \| ... \| \text{Stmt}_l) \\
\text{Stmt} &::= \mathbf{skip} \mid \mathbf{signal} \mid v := \text{PExp} \mid *v := \text{PExp} \mid \text{Stmt}; \text{Stmt} \\
&\quad\mid \mathbf{if}\ \text{BExp}\ \mathbf{then}\ \text{Stmt}\ \mathbf{else}\ \text{Stmt}\ \mathbf{fi} \mid \mathbf{while}\ \text{BExp}\ \mathbf{do}\ \text{Stmt}\ \mathbf{od} \\
&\quad\mid \mathbf{new}(\text{PExp}) \mid \mathbf{del}(\text{PExp}) \mid \langle \text{BExp} : \text{Stmt} \rangle \\
\text{PExp} &::= \mathit{nil} \mid v \mid *v \mid \&v \\
\text{BExp} &::= \text{tt} \mid \text{ff} \mid \text{PExp} = \text{PExp} \mid \text{BExp} \wedge \text{BExp} \mid \neg\text{BExp}
\end{aligned}
$$

$V(\pi) := \{v_1, ..., v_k\}$ denotes the set of variables for $\pi \in \text{LMP}$.

An LM–program thus consists of a declaration of global program variables and a series of statements to be executed in parallel. Each of these statements can either be a pointer assignment, a sequence of statements, a control structure, or a special statement such as **signal** which sets a global signal flag that can be tested in the logic, **new**/**del** for dynamic creation or deletion of objects at runtime (possibly leading to an unbounded number of allocated heap cells) and guarded atomic regions. If the Boolean guard $g$ in $\langle g : s \rangle$ is true, $s$ is executed atomically, i.e., with no interference by other processes. If $g$ is evaluated to false, the process is blocked (until $g$ becomes true).

$$
\begin{array}{l}
\mathbf{var}\ x, y, z( \\
\quad \mathbf{while}\ \text{tt}\ \mathbf{do}\ \langle \text{tt} : \\
\quad\quad \mathbf{if}\ x = \mathit{nil} \\
\quad\quad\quad \mathbf{then}\ \mathbf{new}(y);\ x := y \\
\quad\quad\quad \mathbf{else}\ \mathbf{new}(*y);\ y := *y \\
\quad\quad \mathbf{fi} \\
\quad \rangle\ \mathbf{od} \\
\| \quad \mathbf{while}\ \text{tt}\ \mathbf{do}\ \langle x \neq \mathit{nil} : \\
\quad\quad z := x;\ x := *x;\ \mathbf{del}(z) \\
\quad \rangle\ \mathbf{od} \\
)
\end{array}
$$

Fig. 1. Producer/Consumer

Pointer expressions comprise the special constant *nil* denoting an undefined

pointer value, a program variable, the dereferencing or referencing of a program variable. Note that for simplicity we do not allow arbitrary dereferencing depths; those can be emulated using a sequence of assignments within an atomic region.

**Example 2.1** Figure 1 shows an LM–program implementing a producer inserting objects and a consumer deleting objects at the end (pointed to by $y$) and beginning (pointed to by $x$) of a queue, respectively. If the queue is empty the consumer cannot proceed due to the guard $x \neq nil$ until the producer has inserted at least one object. Insertion and deletion are executed atomically to prevent interferences.

**Definition 2.2** A *heap configuration* of a program $\pi \in$ LMP is a tuple $\gamma = (N, A, \mu, F)$ with a set of nodes $N \supseteq V(\pi)$, a set of abstract nodes $A \subseteq N \setminus PV$, a successor function $\mu : N \to N_{nil}$ (where $N_{nil} := N \cup \{nil\}$), and a set of flags $F \subseteq \{\text{err}, \text{dl}, \text{leak}, \text{signal}, \text{new}\}$.

Let $\mu^\star : 2^N \to 2^N$ with $\mu^\star(X) := \{n \in N \mid \exists k \in \mathbb{N},\ \exists n' \in X : \mu^k(n') = n\}$ be the transitive closure of $\mu$, i.e. all nodes reachable from a node in $X$ (and $X$ itself).

Thus the nodes represent both the dynamic objects created and deleted at runtime and the static program variables (which cannot be deleted). Edges, as formalized by the $\mu$–function, encode the *points–to* information of a specific program state. The set $A$ of abstract nodes will later be used for our abstraction technique and will be empty throughout the current section. Finally the flags give special information about a state, e.g., whether a runtime error or memory leak occurred, a new node was created, or the signal bit has been set using the **signal** command.

To ensure the finiteness of our abstraction we will automatically delete those heap nodes that are not reachable from the program variables. This is accomplished by the following *garbage collection* mapping. Whenever it removes an unreachable node, it sets the leak flag indicating a potential memory leak.

**Definition 2.3** For $\gamma = (N, A, \mu, F)$ we define $\gamma{\downarrow} := (N', A \cap N', \mu \restriction N', F \cup \{\text{leak} \mid (N \setminus N') \neq \emptyset\})$ where $N' = \mu^\star(PV)$.

$\Gamma$ denotes the set of all *garbage–free heap configurations*, i.e., $\forall \gamma \in \Gamma : \gamma{\downarrow} = \gamma$, and $\Gamma_c \subseteq \Gamma$ denotes the set of all concrete configurations, i.e., those with $A_\gamma = \emptyset$.

From now on we will always assume garbage freeness when mentioning heap configurations. This enforces a bound on the maximal number of incoming edges for a node (essentially the number of program variables).

**Definition 2.4** Let $\gamma = (N, \emptyset, \mu, F) \in \Gamma_c$. Then we define the semantics of pointer expressions $[\![\cdot]\!] : \text{PExp} \rightharpoonup N_{nil}$ by[1]:

$$[\![nil]\!] := nil \qquad [\![*v]\!] := \mu([\![v]\!])$$
$$[\![v]\!] := \mu(v) \qquad [\![\&v]\!] := v$$

The semantics of Boolean expressions $[\![\cdot]\!] : \text{BExp} \rightharpoonup \mathbb{B}$ is standard and strict[2]. Note that Def. 2.2 implies that $\mu(nil) = \bot$ and so $[\![\cdot]\!]$ can indeed yield undefined results for both pointer and Boolean expressions.

**Definition 2.5** For $\pi = \text{var } v_1, ..., v_k : (s_1 \| ... \| s_l) \in$ LMP the *concrete operational semantics* is given by a transition system $T^c_\pi = (Q, q_0, lab, \to)$ with a set of states

[1] $\rightharpoonup$ denotes a partial function and $\bot$ the undefined value.

[2] One undefined operand yields an undefined expression.

$Q \subseteq \Gamma_c \times \text{Stmt}_\diamond(\{\|\}\text{Stmt}_\diamond)^\star$ where $\text{Stmt}_\diamond = \text{Stmt} \cup \{\diamond\}\text{Stmt} \cup \{\varepsilon\}$, an initial state $q_0 = ((N_0, \emptyset, \mu_0, \emptyset), s_1\|...\|s_l)$ where $N_0$ and $\mu_0$ represent the "input heap", a labeling $lab : Q \to \Gamma_c$ with $\forall(\gamma, s) \in Q : lab((\gamma, s)) = \gamma$, and a transition relation $\to \subseteq Q \times Q$.

In the following we will use the abbreviations $\hat{F}$ for $F \setminus \{\text{signal}, \text{new}, \text{leak}\}$ and noerr for $\{\text{err}, \text{dl}\} \cap F = \emptyset$. $\gamma_{\text{err}}$ and $\gamma_{\text{dl}}$ will denote pointer error and deadlock states. Most transition rules are straightforward, thus here we will only consider some interesting examples.

$$\frac{[\![ g ]\!] = 1 \qquad \gamma, s \to \gamma', s' \qquad \text{noerr}}{\gamma, \langle g : s\rangle \to \gamma', \diamond s'} \tag{1}$$

$$\frac{\gamma, s \to \gamma', s' \qquad s' \neq \varepsilon \qquad \text{noerr}}{\gamma, \diamond s \to \gamma', \diamond s'} \quad \frac{\gamma, s \to \gamma', \varepsilon \qquad \text{noerr}}{\gamma, \diamond s \to \gamma', \varepsilon} \tag{2}$$

$$\frac{\exists j \text{ s.t. } \gamma, s_j \to \gamma', s_j' \qquad \forall i \neq j : \nexists s_i' \text{ s.t. } s_i = \diamond s_i' \qquad \text{noerr}}{\gamma, s_1\|...\|s_k \to \gamma', s_1\|...\|s_j'\|...\|s_k} \tag{3}$$

$$\frac{\nexists j \text{ s.t. } \gamma, s_j \to \gamma', s_j' \qquad \exists j : s_j \neq \varepsilon \qquad \text{noerr}}{\gamma, s_1\|...\|s_k \to \gamma_{\text{dl}}, \varepsilon} \tag{4}$$

$$\frac{}{\gamma, \varepsilon\|...\|\varepsilon \to \gamma, \varepsilon\|...\|\varepsilon} \tag{5}$$

$$\frac{[\![ \alpha ]\!] \neq \bot \qquad \text{noerr}}{(N, A, \mu, F), v := \alpha \to (N, A, \mu[v/[\![ \alpha ]\!]], \hat{F})\!\downarrow, \varepsilon} \tag{6}$$

$$\frac{\text{noerr}}{(N, A, \mu, F), \mathbf{new}(v) \to (N \uplus \{n_{\text{new}}\}, A, \mu[v/n_{\text{new}}], \hat{F} \cup \{\text{new}\})\!\downarrow, \varepsilon} \tag{7}$$

$$\frac{[\![ \alpha ]\!] \in N \setminus PV \qquad \text{noerr}}{(N, A, \mu, F), \mathbf{del}(\alpha) \to (N \setminus \{[\![ \alpha ]\!]\}, A, \mu[[\![ \alpha ]\!]/\bot, \mu^{-1}([\![ \alpha ]\!])/nil], \hat{F})\!\downarrow, \varepsilon} \tag{8}$$

Some remarks on the transition rules are in order. The leak, signal, and new flags are reset after each transition; they are only activated in the state directly following the corresponding "event".

Regarding the concurrency rules we need to take care of the special semantics of the atomic regions. If a process is executing such a statement it must not be interrupted, and therefore the corresponding state is marked with $\diamond$ (rule 1). The interleaving rule 3 excludes that any other than process $j$ is in an atomic region. If no process can proceed (all are blocked) we reach the special deadlock state (rule 4). If all processes are terminated or an error or deadlock state is reached the program loops to ensure that all paths in the transition system are infinite (rule 5).

The treatment of assignments (rule 6) and the **new** statement (rule 7) is again straightforward, we though have to keep in mind in the first case that runtime errors might occur (dereferencing of *nil* pointers) and that garbage may be generated. Rule 8 handles the deletion of nodes. Please note that the next–pointers of the predecessors of the deleted node are set to *nil* (mainly to avoid case distinctions for undefined expressions in the semantics).

We conclude that for the producer/consumer example (Fig. 1) the state space

becomes infinite when applying the operational semantics as defined above.

# 3 State–Space Abstraction

As we have seen in the previous section the state space of LM–programs can get infinite even for simple example programs making standard verification methods inapplicable. To tackle the problem we use abstraction techniques to generate an *abstract transition system* that incorporates the behavior of the concrete one, i.e., whose runs cover all concrete ones. This approach is correct but generally incomplete: although we can conclude from the satisfaction of a property in the abstract state space its validity in the concrete case, the inverse is impossible though. But since the abstraction is parameterized via a global constant $M \in \mathbb{N}$ we can refine the abstraction depending on our needs. For a given $M > 0$ we set $\mathbb{M} := \{0, 1, ..., M, \star\}$, where $\star$ represents all values greater than $M$.

**Chain Abstraction**

The main idea of our abstraction is to summarize subgraphs of a configuration into summary nodes [45], which will be exactly those contained in the $A$–component of a heap configuration. Summary nodes (also called abstract nodes) are not allowed to represent arbitrary structures but only so–called *chains*, i.e., non–interrupted lists. Our abstraction is based on [18,19] with the difference that nodes are either truly abstract or concrete, thus recording node multiplicities is not necessary.

**Definition 3.1** Let $\gamma = (N, A, \mu, F) \in \Gamma$ be a configuration. A nonempty set of nodes $C \subseteq N$ is called a *chain* if either

- $|C| = 1$ and $C \subseteq PV$ or
- $C \cap PV = \emptyset$ and there exists a bijection $\pi : \{1, ..., |C|\} \to C$ such that $\mu(\pi(i)) = \pi(i + 1)$ for $i \in \{1, ..., |C|\}$ and $\forall i \in \{2, ..., |C|\} : |\mu^{-1}(\pi(i))| = 1$.

For a given chain $C$ we will use the abbreviations $\overleftarrow{C} := \pi(1)$, and $\overrightarrow{C} := \pi(|C|)$. A chain is called *maximal* if no superset $C' \supset C$ is a chain.

Thus a chain is a sequence of pointer–connected nodes without interference of other incoming edges or a singleton set containing a program variable. It follows that the abstraction of chains preserves the graph structure. We will now introduce a type of functions, called *abstraction morphisms*, that is based on this concept.

**Definition 3.2** Let $\gamma_i = (N_i, A_i, \mu_i, F_i) \in \Gamma,\ i \in \{1, 2\}$ be two heap configurations. An *abstraction morphism* $h : N_1 \to N_2$ satisfies for all $v \in PV \cap N_1$ and $n_i, n_i' \in N_i$:

1. $h(v) = v$
2. $h^{-1}(n_2)$ is a chain in $N_1$
3. $\mu_2(n_2) = n_2' \ \Rightarrow\ \mu_1(\overrightarrow{h^{-1}(n_2)}) = \overleftarrow{h^{-1}(n_2')}$
4. $\mu_1(n_1) = n_1' \Rightarrow h(n_1) = h(n_1') \vee \mu_2(h(n_1)) = h(n_1')$
5. $n_2 \in A_2 \ \Leftrightarrow\ h^{-1}(n_2) \cap A_1 \neq \emptyset \ \vee\ |h^{-1}(n_2)| > M$
6. $F_1 = F_2$

We write $h : \gamma_1 \rightarrowtail \gamma_2$ to denote that the abstraction morphism $h$ abstracts $\gamma_1$ to $\gamma_2$ and $\gamma_2 \leq \gamma_1 \ \Leftrightarrow\ \exists h : \gamma_1 \rightarrowtail \gamma_2$.

Abstraction morphisms abstract from concrete chains with minimal length $M+1$ (cond. 2 and 5). The preservation of the graph structure is enforced by conditions 3 and 4. Program variables, being special nodes, remain untouched (cond. 1).

**Example 3.3** Figure 2 shows an abstraction morphism for $M = 1$. The dashed lines represent the mapping, and the black nodes denote the resulting abstract nodes. Note that for $M = 2$ the nodes 3 and 4 could not be projected onto the same abstract node (condition 5 of Def. 3.2). The chain $\{3, 4\}$ cannot be extended by node 5, since this node has two incoming edges which is only allowed for the first node of a chain. Although in this example the source configuration is concrete, this is of course not necessary by definition.

An important property of abstraction morphisms is their surjectivity. If, in addition a morphism is injective it becomes an isomorphism. Isomorphic configurations cannot be distinguished except for node naming, the graph structure is the same.

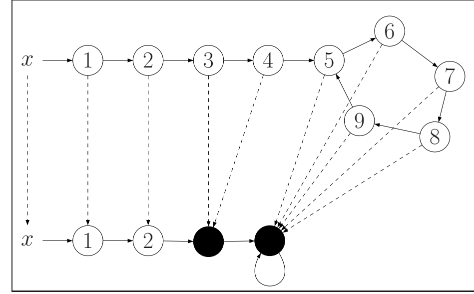



Fig. 2. An Abstraction Morphism

### Canonical Configurations

Previously we have defined how configurations can be abstracted. It remains the problem that there can be different abstractions of a given source configuration. For this reason we need a normal form that implies uniqueness. In the following we define this normal form, assuming $\gamma = (N, A, \mu, F) \in \Gamma$.

**Definition 3.4**

(i) Let $\lfloor N \rfloor_j := \{n \in N \mid \nexists v \in PV : \mu^k(v) = n,\ k < j\}$ be the set of nodes with a distance of at least $j$ from the variable nodes. Analogously $\lceil N \rceil_j := N \setminus \lfloor N \rfloor_{j+1}$.

(ii) A configuration $\gamma$ is called *canonical* if $\lceil N \rceil_2 \cap A = \emptyset$ and for all maximal [3] chains $C \subseteq \lfloor N \rfloor_3$ either $|C| = 1$ or $|C| \leq M \ \wedge \ C \cap A = \emptyset$. The set of all canonical configurations is denoted by $\Gamma_\natural$.

The notion of canonical configurations is quite intuitive: maximal chains are collapsed where possible but only up to a distance of three from variable nodes. The latter condition ensures that pointer expressions always evaluate to concrete nodes, which will simplify the definition of the abstract LMP semantics. The abstraction morphism in Fig. 2 yields a canonical configuration, as can be easily verified.

**Theorem 3.5 (Existence)** *For every $\gamma \in \Gamma$ with $\lceil N \rceil_2 \cap A = \emptyset$ there exists a $\gamma' \in \Gamma_\natural$ such that $\gamma' \leq \gamma$.*

It is easy to construct a morphism yielding a canonical configuration. It has to collapse maximal chains that are larger than $M$ or contain abstract nodes, if they are sufficiently distant from the variable nodes. In the following we will call this morphism $h_\natural$. The precise definition does not matter as states the following theorem.

[3] Here we refer to maximality in $\lfloor N \rfloor_3$.

**Theorem 3.6 (Uniqueness)** *Let $\gamma \in \Gamma$ and $\gamma_1, \gamma_2 \in \Gamma_\natural$ such that $h_1 : \gamma \rightarrowtail \gamma_1$ and $h_2 : \gamma \rightarrowtail \gamma_2$ are two abstraction morphisms. Then $\gamma_1$ and $\gamma_2$ are isomorphic.*

The proof of the uniqueness had to be omitted here. The consequence of these results is the appropriateness of canonical configurations as a normal form. The abstract semantics will operate on such configurations.

**Abstract Semantics of List–Manipulating Programs**

As already mentioned, our goal is to guarantee the correctness of our abstraction approach. This can be achieved by ensuring that every concrete execution of a given system can be "simulated" by an abstract computation, which necessarily introduces nondeterministic behavior on the abstract side.

Regarding the expression semantics nothing needs to be modified: in a canonical configuration, abstract nodes have a distance greater than two from the variable nodes such that every pointer expression refers to a concrete node. The expression semantics can therefore be chosen identical to the concrete case (Def. 2.4), now interpreted on canonical configurations.

**Definition 3.7** Given a program $\pi = \mathbf{var}\ v_1, ..., v_k : (s_1 \| ... \| s_l) \in \text{LMP}$, its *abstract operational semantics* is defined by the labeled transition system $T_\pi^a = (Q, [q_0]_\cong, lab, \rightarrow)$ with state set $Q \subseteq \Gamma_\natural/_\cong \times \text{Stmt}_\diamond(\{\|\}\text{Stmt}_\diamond)^\star$, initial state $q_0$ as in Def. 2.5, labeling function $lab : Q \rightarrow \Gamma_\natural$ where $\forall (K, s) \in Q : lab((K, s)) = K$, and transition relation $\rightarrow$ as specified by the following rules (we focus on the assignments, since the other rules are analogous to the concrete case, but operating on isomorphism congruence classes).

$$\frac{\alpha \notin *V(\pi) \qquad \text{noerr}}{[(N, A, \mu, F)]_\cong, v := \alpha \rightarrow [h_\natural((N, A, \mu[v/[\![\alpha]\!]], \hat{F})\!\downarrow)]_\cong, \varepsilon} \tag{1}$$

$$\frac{\gamma' \in \Gamma_\natural \text{ s.t. } h_\natural((N, A, \mu[v/[\![*w]\!]], \hat{F})\!\downarrow) \leq \gamma' \qquad [\![w]\!] \neq nil \qquad \text{noerr}}{[(N, A, \mu, F)]_\cong, v := *w \rightarrow [\gamma']_\cong, \varepsilon} \tag{2}$$

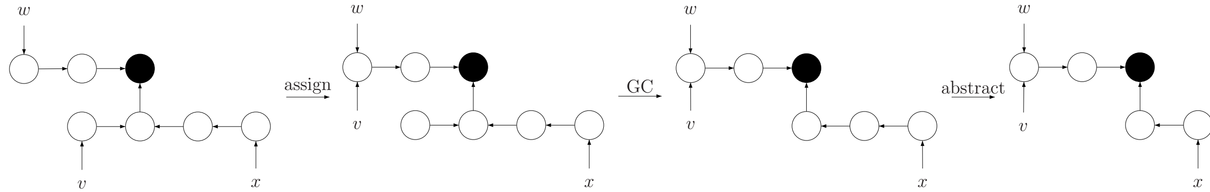
$$\frac{[\![v]\!] \neq nil \qquad [\![\alpha]\!] \neq \bot \qquad \text{noerr}}{[(N, A, \mu, F)]_\cong, *v := \alpha \rightarrow [h_\natural((N, A, \mu[\mu(v)/[\![\alpha]\!]], \hat{F})\!\downarrow)]_\cong, \varepsilon} \tag{3}$$

$$\frac{[\![\alpha]\!] = \bot \ \vee [\![\alpha']\!] = \bot \qquad \text{noerr}}{[\gamma]_\cong, \alpha := \alpha' \rightarrow [\gamma_{\text{err}}]_\cong, \varepsilon} \tag{4}$$

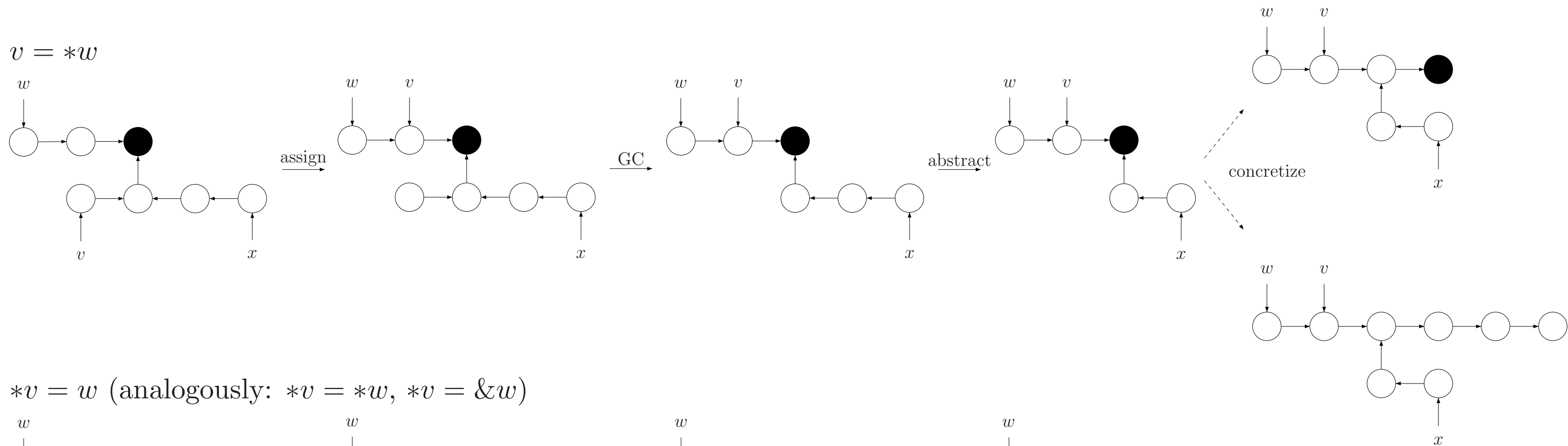
In Fig. 3 the semantic rules are visualized for an example configuration. In rule 2 there might be the necessity for both abstraction and concretion. The execution of the assignment and the following abstraction via $h_\natural$ yields an intermediate configuration which is generally not canonical since the variable $v$ could now be too close to an abstract node. Therefore we have to find a canonical configuration $\gamma'$ that is at least as concrete as $\bar{\gamma}$ and related by an abstraction morphism to it. There might be more than one solution, thus this rule is nondeterministic (indicated by the dashed arrows), but remains the only source of nondeterminism.

In rules 1 and 3 the distance to an abstract node is not reduced, but the opposite case can occur: just imagine an assignment of the form $y := nil$. If $y$ points into a list whose head is referred to by another variable, we possibly increase the distance from that variable to abstract nodes. The execution of the assignment

(1) $v = w$ (analogously: $v = nil$, $v = \&w$)




(2) $v = *w$




(3) $*v = w$ (analogously: $*v = *w$, $*v = \&w$)

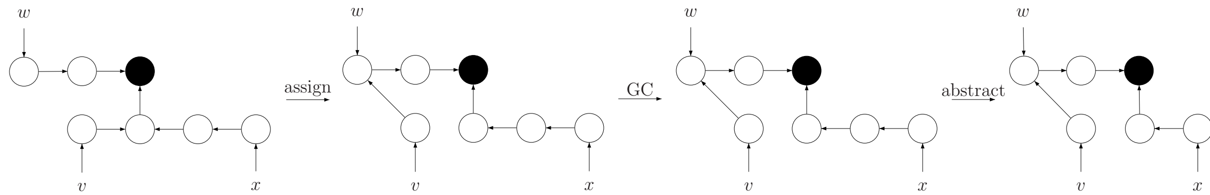



Fig. 3. Exemplary visualization of the abstract semantics ($M = 3$)

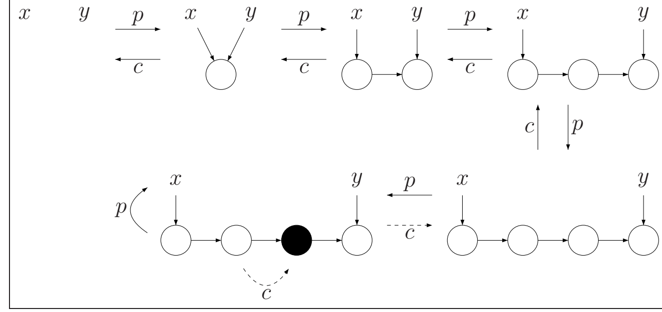



Fig. 4. Producer/Consumer: Abstract State Space ($M = 1$)

therefore potentially yields a non–canonical configuration and we have to re–abstract to determine the corresponding canonical configuration. According to Thm. 3.6 the result is unique and thus these steps are deterministic.

**Example 3.8** Figure 4 shows the finite abstract state space of the producer/consumer program from Fig. 1 for $M = 1$. The $p$– and $c$–transitions each summarize several producer/consumer steps. The dashed transitions are nondeterministic steps, since the abstract node, visualized in black color, represents at least two nodes in a chain. If now the consumer deletes one node from the beginning of the queue the distance of $x$ to the abstract node becomes two and thus we need to *concretize* the graph to obtain a canonical configuration. For this we distinguish two cases: either the abstract node represents exactly two nodes, then we reach the graph to the right, or it represents more than two, in which case we stay in the same state since the abstract node still represents more than one concrete node.

**Theorem 3.9 (Finiteness)** *For every* $\pi \in \mathrm{LMP}$, $T^a_\pi$ *is finite.*

The idea of the proof is to establish a bound on the number of nodes of canonical configurations for a given number of program variables.

**Theorem 3.10 (Correctness of the Abstraction)** *Let* $\pi \in \mathrm{LMP}$. *For every transition in* $T^c_\pi$ *there exists a corresponding abstract transition in* $T^a_\pi$ *such that the heaps are related by abstraction morphisms.*

The proof of the correctness theorem has been omitted due to space constraints.

# 4 A Logic for Concurrent List–Manipulating Programs

In the previous sections we have defined our programming language for concurrent pointer manipulation and both its concrete and abstract semantics. In this section we will present a logic which will allow us to reason about heap configurations and program behavior. In the following $LV$ denotes a set of logical variables, where we always assume that $LV \cap PV = \emptyset$.

**Pointer Logic**

Pointer logic deals with single configurations and is employed to express graph properties as well as to inspect the special flags of heap configurations (see Def. 2.2).

**Definition 4.1** The set PL of *Pointer Logic formulas* is given by the grammar

$$\begin{array}{lll} \text{NExp} & ::= & \mathit{nil} \mid v\ (\in PV) \mid x\ (\in LV) \mid *\text{NExp} \\ \text{Atomic} & ::= & \text{tt} \mid \text{ff} \mid \text{err} \mid \text{dl} \mid \text{leak} \mid \text{signal} \mid \text{new} \mid \text{NExp} = \text{NExp} \mid \text{NExp} \leadsto \text{NExp} \\ \text{PL} & ::= & \text{Atomic} \mid \neg\text{PL} \mid \text{PL} \wedge \text{PL} \mid \exists x : \text{PL} \end{array}$$

Later on we will use the logical operations $\vee$, $\rightarrow$, $\leftrightarrow$, and $\forall$ (defined as usual) as abbreviations. Note that in contrast to pointer expressions in LM–programs our logic supports dereferencing operations of arbitrary depth. The special operation $\alpha \leadsto \alpha'$ expresses the reachability of heap objects.

**Definition 4.2** Let $\beta : LV \rightharpoonup N$ be a variable valuation and $\gamma \in \Gamma_c$ a concrete heap configuration. Then we define $[\![\cdot]\!] : \text{NExp} \rightharpoonup N_{nil}$ by:

$$\begin{array}{rlrl} [\![\mathit{nil}]\!] & := \mathit{nil} & [\![v]\!] & := v \text{ for } v \in PV \\ [\![x]\!] & := \beta(x) \text{ for } x \in LV & [\![*\alpha]\!] & := \mu_\gamma([\![\alpha]\!]) \text{ for } \alpha \in \text{NExp} \end{array}$$

Note the semantic difference with respect to the programming language. In navigation expressions a variable $v$ is interpreted by itself and not by the node it is referencing. This avoids the necessity of the referencing operator &.

**Definition 4.3** The (concrete) satisfaction relation $\models$ for PL–formulas is given as follows [4] (for $\gamma = (N, \emptyset, \mu, F)$):

$$\begin{array}{ll} \gamma, \beta \models f & \text{iff } f \in F \text{, where } f \in \{\text{err}, \text{dl}, \text{leak}, \text{signal}, \text{new}\} \\ \gamma, \beta \models \alpha_1 = \alpha_2 & \text{iff } [\![\alpha_1]\!] = [\![\alpha_2]\!] \neq \bot \\ \gamma, \beta \models \alpha_1 \leadsto \alpha_2 & \text{iff } [\![\alpha_i]\!] \neq \bot \ \wedge\ [\![\alpha_2]\!] \in \mu^\star([\![\alpha_1]\!]) \\ \gamma, \beta \models \exists x : \varphi & \text{iff } \exists n \in N : \gamma, \beta[x/n] \models \varphi \end{array}$$

**Temporal Pointer Logic**

Pointer Logic enables us to express properties of single configurations. However it cannot be used to specify (ongoing) computations, i.e., configuration sequences. To this aim we will now extend this logic by temporal operators.

**Definition 4.4** The set TPL of *Temporal Pointer Logic formulas* is given as follows:

$$\text{TPL} ::= \text{PL} \mid \neg\text{TPL} \mid \text{TPL} \wedge \text{TPL} \mid \mathbf{X}\ \text{TPL} \mid \text{TPL}\ \mathbf{U}\ \text{TPL}$$

For $\varphi \in \text{TPL}$ we use the abbreviations $\mathbf{F}\varphi := \text{tt}\mathbf{U}\varphi$ and $\mathbf{G}\varphi := \neg\mathbf{F}\neg\varphi$. Moreover $V(\varphi) \subseteq LV$ denotes the set of (bound or free) logical variables occurring in $\varphi$.

Note that it is *not* possible to nest quantifiers and temporal operators. To do so it would be necessary to keep track of the object identities between states, which is difficult in the presence of abstract nodes. In addition it would blow up the state space and exclude the use of standard model checking algorithms. To the best of our knowledge the only approach to support this idea is the one in [18,19,20]; other works in the area such as [46] consider only *shapes* of the heap. This results in a loss

[4] For $\wedge, \neg$, tt and ff the semantics is standard and therefore omitted.

of expressivity, e.g., a property like $\forall x : \text{new}(x) \to \mathbf{F}\ \text{del}(x)$ which states that every produced object will eventually be consumed cannot be formulated. Nonetheless we can specify many interesting properties.

**Example 4.5** For our producer/consumer system from Fig. 1 it holds true:

1. $\neg\mathbf{F}(\text{dl} \vee \text{err})$ (never deadlock or pointer errors)
2. $\mathbf{GF}\ \text{new}$ (new objects are created infinitely often)
3. $\mathbf{G}((*x \neq nil \vee *y \neq nil) \to (x \rightsquigarrow *y \wedge \forall v : (v \neq y \to x \rightsquigarrow v)))$
   (whenever the queue is not empty, the object $y$ points to is reachable from $x$ and between $x$ and this object lies a chain)

More general correctness properties are:

4. $\mathbf{F}\ *x = *y$ ($x$ and $y$ will eventually become aliases)
5. $\mathbf{G}\neg(\exists z : (x \rightsquigarrow z \wedge y \rightsquigarrow z))$ ($x$ and $y$ always point to disjoint parts of the heap)
6. $\mathbf{G}(\forall y : (x \rightsquigarrow y \to (\neg\exists z : (y \rightsquigarrow z \wedge *z \rightsquigarrow y))))$
   ($x$ always points to a non–cyclic list)
7. $\mathbf{FG}(\neg\text{leak})$ (only finitely often a memory leak can occur)
8. $\mathbf{G}(\forall y : (x \rightsquigarrow y \to (\forall z : (z \rightsquigarrow y \to x \rightsquigarrow z))))$ ($x$ always points to a chain)

As mentioned before, TPL specifies computation paths. The set of possible paths is represented by a transition system.

**Definition 4.6** Let $T = (Q, q_0, lab, \to)$ be a (concrete) transition system with $lab : Q \to \Gamma_c$. A *path* in $T$ is an infinite sequence of states $\rho = \rho_0\rho_1\rho_2... \in Q^\omega$ such that $\rho_i \to \rho_{i+1}$ for all $i \in \mathbb{N}$. Then for $\varphi \in PL$ we have

$$\rho \models \varphi\ (\in \text{PL}) \text{ iff } \exists \beta : LV \rightharpoonup N_{lab(\rho_0)} \text{ s.t. } lab(\rho_0), \beta \models_{\text{PL}} \varphi$$

For the temporal operators the semantics is identical to the one of LTL. We write $T \models \varphi$ iff $\rho \models \varphi$ for all paths $\rho \in \{q_0\}Q^\omega$ in $T$.

### Reasoning about Abstract Computations

As expected the concrete semantics is straightforward. When we switch to abstract configurations, however, we run into several complications since logical variables can be bound to both concrete and abstract nodes. In the latter case we have to record *which* concrete node, represented by the summary node, it is bound to. This could lead to undefinedness of Pointer Logic formulas. This problem occurs mainly in direct comparisons of the form $\alpha = \alpha'$. To tackle this problem we choose the global precision constant $M$ in dependence of the formula as follows. If $\varphi \in \text{TPL}$ is the formula to check, then we assume from now on that

$$M \geq \sum_{x \in V(\varphi)} \{j + 1 \mid *^j x \text{ occurs in } \varphi\}.$$

Due to the presence of abstract nodes it is not sufficient anymore to evaluate logical variables by simple variable–to–node mappings. Additionally we must record the offset of a variable referring to an abstract node and the distance between variables pointing to the same abstract node. This leads to the concept of *abstract valuations*.

Given $\gamma \in \Gamma_\natural$ and $\varphi \in \mathrm{TPL}$, an *abstract valuation* is of the form $\eta = (\beta, o, \delta)$, where $\beta : V(\varphi) \to N_\gamma$ maps logical variables to (abstract) nodes, $o : V(\varphi) \to \mathbb{M}$ denotes the offset for an abstract node, and $\delta : V(\varphi) \to V(\varphi) \rightharpoonup \mathbb{M}$ is a "distance matrix" for the logical variables with potentially undefined entries. $\delta$ is only defined if both arguments are mapped onto the same entity, and $o$ is only different from 1 if the corresponding variable is mapped onto an abstract node. The set of all such valuations will be denoted by $\mathrm{Val}_{\gamma,\varphi}$.

Using this concept one can define a function $d_{\gamma,\eta} : \mathrm{NExp} \times \mathrm{NExp} \to \{0, 1, \infty\}$ measuring the "distance" of pointer expressions, where distance here means either 0 if the expressions are mapped onto the same (concrete) entity, 1 if the the first case does not hold but the second argument is reachable from the first or $\infty$ if neither is the case.

The presence of abstract nodes plays a vital role in the abstract semantics. Without the global constraint for $M$ we would not be able to resolve all possible cases of abstract valuations, a third truth value would thus become necessary. The distance function $\delta$ is required for the case that both variables are mapped onto an abstract node with offset $\star$. With the help of the distance function the abstract semantics of PL and TPL is straightforward.

**Definition 4.7** Let $\gamma = (N, A, \mu, F) \in \Gamma_\natural$ and $\eta = (\beta, o, \delta) \in \mathrm{Val}_{\gamma,\varphi}$. The satisfaction relation $\models$ for PL–formulas on canonical configurations is then given as follows (omitting the trivial cases):

$$
\begin{array}{lll}
\gamma, \eta \models f & \text{iff } f \in F, \text{ where } f \in \{\mathrm{err}, \mathrm{dl}, \mathrm{leak}, \mathrm{signal}, \mathrm{new}\} \\
\gamma, \eta \models \alpha_1 = \alpha_2 & \text{iff } d_{\gamma,\eta}(\alpha_1, \alpha_2) = 0 \\
\gamma, \eta \models \alpha_1 \leadsto \alpha_2 & \text{iff } d_{\gamma,\eta}(\alpha_1, \alpha_2) \in \{0, 1\} \\
\gamma, \eta \models \exists x : \varphi & \text{iff } \exists n \in N,\ \mathit{off} \in \mathbb{M},\ \mathit{dist} : V(\varphi) \rightharpoonup \mathbb{M} \text{ s.t.} \\
& \qquad \gamma, (\beta_\eta[x/n], o_\eta[x/\mathit{off}], \delta_\eta[x/\mathit{dist}]) \models \varphi
\end{array}
$$

Let $T = (Q, q_0, \mathit{lab}, \to)$ be an abstract transition system with $\mathit{lab} : Q \to \Gamma_\natural/\cong$ and $\rho \in Q^\omega$ a path in it. Then $\rho \models \varphi \in \mathrm{PL}$ iff for $\gamma \in \mathit{lab}(\rho_0)$ there exists an $\eta \in \mathrm{Val}_{\gamma,\varphi}$ s.t. $\gamma, \eta \models_{\mathrm{PL}} \varphi$. Temporal operators and Boolean connectives are treated in the standard way. We write $T \models \varphi$ iff $\rho \models \varphi$ for all paths $\rho \in \{q_0\}Q^\omega$ in $T$.

The following theorem states that the abstract semantics of TPL and of the programming language is correct, i.e., that the validity of a formula under the abstract interpretation implies the validity under the concrete one. The converse though does not hold.

**Theorem 4.8** *Let* $\pi \in \mathrm{LMP}$ *and* $\varphi \in \mathrm{TPL}$. *If* $T^a_\pi \models \varphi$ *then* $T^c_\pi \models \varphi$.

**Proof.** It suffices to show for all $\varphi \in \mathrm{PL}$ and $\gamma \in \Gamma_c$ the proposition:

$$\exists \beta : LV \rightharpoonup N_\gamma \text{ s.t. } \gamma, \beta \models \varphi \;\Leftrightarrow\; \exists \eta \in \mathrm{Val}_{\gamma,\varphi} \text{ s.t. } h_\natural(\gamma), \eta \models \varphi \qquad (\star)$$

Note that the $\Leftarrow$–direction is sufficient for correctness, the $\Rightarrow$–direction though is trivial. In the proof the choice of the global constant $M$ (depending on the formula) plays a central role. Imagine for example a property "the heap contains at least five objects different from program variables". To formulate this property we need at least five different logical variables and the constraint on $M$ implies that $M \geq 5$. For smaller $M$ it can happen that a formula that is satisfied in the abstract case, does

not hold in all concrete configurations associated with the abstract one. E.g. for $M = 1$ and a graph with one abstract node our example property would be satisfied; in the corresponding concrete graph where the abstract node is represented by two concrete nodes not necessarily.

With ($\star$) we can infer from Thm. 3.10 the validity of the claim, since TPL does not allow path quantifiers. By construction of the abstract PL–semantics it is intuitively clear that ($\star$) holds. □

#### Model Checking Temporal Pointer Logic

Because of the two–stage approach in defining the logic, we can reduce the TPL model checking problem to an LTL model checking problem, which can efficiently be verified by existing model checkers.

**Algorithm 1** *Let* $T = (Q, q_0, lab, \rightarrow)$ *be the abstract transition system generated by a program* $\pi \in \mathrm{LMP}$ *and* $\varphi \in \mathrm{TPL}$ *the formula to verify. Let* $\Psi := \{\psi \in \mathrm{PL} \mid \psi$ *maximal subformula of* $\varphi\} = \{\psi_1, ..., \psi_r\}$.

*Define a "traditional" transition system* $T' = (Q, q_0, lab', \rightarrow)$ *where* $lab' : Q \rightarrow 2^{AP}$ *with* $AP = \{p_i \mid i \in \{1, ..., r\}\}$ *such that* $p_i \in lab'(q) \ \Leftrightarrow \ lab(q) \models \psi_i$.

*Now solve the LTL model checking problem* $T' \models^{?}_{\mathrm{LTL}} \varphi[\psi_1/p_1, ..., \psi_r/p_r]$.

The idea is thus to replace all (maximal) PL–subformulas by atomic propositions to obtain an LTL–formula. To do so we first have to evaluate the PL–formulas on the transition system and to change its labeling from configurations to atomic propositions, where each atomic proposition represents the truth value of the corresponding PL–subformula on the given configuration. The correctness of this approach is clear.

#### Limitations

Due to the nondeterminism in the abstract semantics caused by the presence of abstract nodes we may obtain *false negatives*. This means that in the abstract transition system there may exist computations which do not correspond to concrete ones and on which the property to verify does not hold.

Consider a program creating a list (pointed to by $v$) with $M + 3$ elements and then deleting again $M + 3$ elements. The property to verify is $\mathbf{XF}(*v = nil)$, i.e. that the list becomes empty. It is obvious that due to the presence of an abstract node after the construction of the list in the abstract semantics there is a path that retains that abstract node and thus the list never becomes empty (see Def. 3.7, rule 2). In the concrete case however the formula is satisfied.

Due to the overapproximation and the LTL approach false *positives* though cannot occur. This means that the successful verification of a property in the abstract case implies the correctness in the concrete case. False negatives can only occur in cases where information on the precise number of objects is necessary.

## 5 Application: Concurrent Garbage Collection

In this section we will show we will employ our approach to find counterexamples of a concurrent garbage collection algorithm. More concretely we will consider a so–called *mark–and–sweep* collector, which maintains a bit for each object in the heap to record its reachability status. Here we model this information as an additional heap component, a (partial) function $r : N \rightharpoonup \mathbb{B}$ which indicates whether the

collector considers a node to be reachable (1) or not (0). This component is made accessible to the garbage collector program using the additional constructs

- **reset** $\in$ Stmt, which resets the reachability value of every node to 0,
- **mark**$(\alpha) \in$ Stmt where $\alpha \in$ PExp, which sets the reachability information of the node $[\![\alpha]\!]$ to 1, and
- $r(\alpha) \in$ BExp where $\alpha \in$ PExp tests whether the reachability bit of $[\![\alpha]\!]$ is set.

We refrain from giving the formal details of the extended syntax and semantics of LM–programs; these are straightforward to formalize. The only modification we would like to mention explicitly is an adaptation of the automatic garbage collection procedure (cf. Def. 2.3), which is activated after the execution of every LM–statement which potentially causes nodes to become unreachable (we refer to the derivation rules in Def. 2.5). To ensure the finiteness of our abstraction, we still have to use it. However, we will adapt the handling of the leak flag such that it will be set only if the garbage collector considers an unreachable node $n$ to be reachable, i.e., if $r(n) = 1$. Formally this means that for an extended configuration $\hat{\gamma} = (N, A, \mu, F, r)$ we define $\hat{\gamma}{\downarrow} := (N', A \cap N', \mu \restriction N', F \cup \{\text{leak} \mid \exists n \in (N \setminus N') : r(n) = 1\}, r \restriction N')$ with $N' = \mu^\star(PV)$.

Using these concepts we can now proceed by describing how a concurrent garbage collector can be added to a given LM–program, called a *mutator*. For $\pi =$ **var** $v_1, \ldots, v_k : (s_1 \| \ldots \| s_l) \in$ LMP, we define $\pi' :=$ **var** $v_1, \ldots, v_k, t : (s_1 \| \ldots \| s_l \| c)$ with garbage collector $c$ as in Fig. 5.

Thus the garbage collector is running concurrently with the mutator. It executes an infinite loop, starting by resetting the reachability bit of every node in the heap. Using the auxiliary variable $t$, it then marks every reachable node, beginning with the roots of the heap which are accessible by the program variables. Here the statement **with** $v \in PV$ **do** $s$ **od** is a meta construct which is expanded to $s[v/v_1]; s[v/v_2]; ...; s[v/v_k]$ for $PV = \{v_1, \ldots, v_k\}$. Whenever it encounters a node which has already been marked (**if** statement), it continues with the next program variable to avoid redundant assignments. Finally it employs the signaling mechanism of our programming language to indicate that now the actual collection phase would start, i.e., that all nodes whose reachability bit is 0 would be removed.

```
while tt do
  reset;
  with v ∈ PV do
    t := v;
    while t ≠ nil do
      if r(t) then t := nil
            else  mark(t);
                  t := *t
      fi
    od
  od;
  signal
od
```

Fig. 5. A naive garbage collector

Note, however, that we are still using our automatic garbage collection procedure such that we can guarantee that in every configuration of the system, all nodes are reachable. In other words, whenever the signal occurs there should not exist any unmarked node in the heap. This observation is the key idea for specifying the *soundness* of the garbage collector $c$ as a safety property in TPL. Here we assume that the underlying Pointer Logic (cf. Def. 4.1) is extended by atomic propositions of the form $r(\alpha)$ which allow us test the reachability information of the node to which the navigation expression $\alpha$ refers:

$$\mathbf{G}(\text{signal} \rightarrow \forall x : r(x))$$

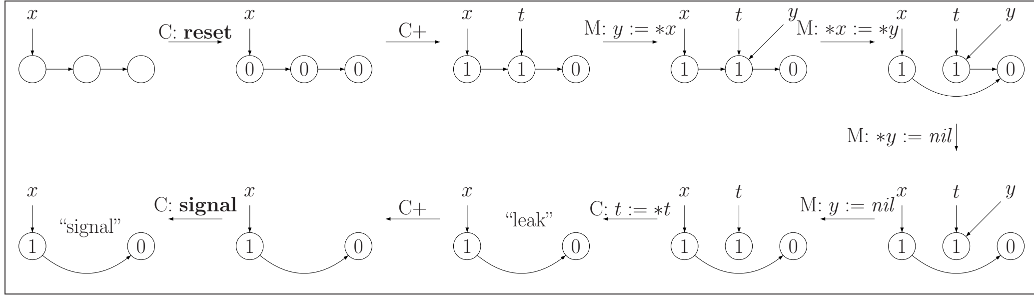



Fig. 6. Possible erroneous run of garbage collector and mutator

Another important issue is the *completeness* of the garbage collector, which means that every node which has become unreachable in the course of the computation, will eventually be removed. This, however, cannot be directly expressed for two reasons. First, verifying this property would require to keep track of the identity of objects between different configurations, which in turn involves the nesting of quantifiers and temporal operators. This is not supported by our logic. Second, our automatic garbage collection procedure immediately removes nodes that have become unreachable.

What we can formulate instead, however, is a safety property which comes very close to the actual completeness. It expresses that a node which has become unreachable will never be marked by the garbage collector. Employing the modified handling of the leak flag, this property can simply be formulated as

$$\mathbf{G}\ \neg\text{leak}$$

Note that this formalization is only justified since the garbage collector is monotonic in the following sense: once a node has been marked, its reachability information will not be reset before the collection signal occurs. Moreover completeness can only be expected (just as the above soundness property) if it is guaranteed that the mutator does not modify the reachability bits.

The example computation in Fig. 6 shows that the above garbage collector violates both of these requirements. Here the mutator program is assumed to be of the form $y := *x;\ *x := *y;\ *y := nil;\ y := nil$; it simply discards the second node of the list whose head is referenced by $x$ (assuming that this node exists). Here C and M stand for operations of the collector and the mutator, respectively, which are either concretely given or summarized by a "+" sign. The bits labeling the nodes indicate the reachability information as set by the collector.

The computation shows that the collector is neither correct nor complete. In the final step involving the signal flag, the reachability value 0 of the list's tail node means that it would be removed by the collector although it is reachable. Two steps earlier, the leak flag indicates that garbage has automatically been deleted which has been marked as reachable by the collector. Both of these problems are caused by the uncontrolled interaction between the mutator and the collector; they can be avoided by placing the body of the collector loop in an atomic region.

# 6 Related Work

Related work on the topic of analyzing pointer–manipulating programs can be classified into the following (often overlapping) categories.

**Predicate abstraction** abstracts the state space of the program by evaluating it under a number of given predicates. This yields a Boolean program which conservatively simulates all potential executions [25]. Successful software model checkers such as BLAST [28] and SLAM [3] are based on this approach. There are several papers that use classical predicate abstraction for pointer analysis [2,14]. In particular, [15,16] study concurrent garbage collection using predicate abstraction.

**Shape analysis** is a static analysis framework that represents recursive data structures of unbounded size by finite structures, called "shape graphs". The idea is to apply to the heap the same abstraction that is applied to the program's states in predicate abstraction: it is defined in terms of equivalence classes of heap objects that are induced by a finite set of predicates on those objects. The usual approach is to formalize shape graphs by three–valued logical structures [46]. This approach has been implemented TVLA [34] and in BLAST [5] which makes use of TVLA.

Recent developments comprise the development of adaptive methods which automatically adjust to the data structures that occur in the given program [31,35,48], demand–driven techniques [5,27], efficiency improvements [33], and interprocedural shape analysis [26,30,43,44].

It is often argued that the application of predicate abstraction to pointer structures does not work well because it is difficult to find predicates which abstract heap structures in an appropriate and compact way [5]. This claim is substantiated by the results in [36] which investigates the application of both predicate abstraction and shape analysis to programs operating on singly–linked lists, employing a similar abstraction as ours: elements on unshared list segments are summarized. It is shown that standard predicate abstraction requires an exponential number of predicates in comparison to the number of predicates in shape analysis. Also [41] considers both techniques, but in a very restricted programming–language setting which only supports single assignments.

**Regular model checking** is a framework for unified verification of infinite–state systems based on automata theory. It represents states using words (trees) over a finite alphabet and sets of states using finite (tree) automata [10]. Like in our approach, singly–linked lists are also considered in [8,9], but only safety and termination properties are verified.

**Dataflow analysis** is a technique for gathering information about certain aspects of a program using its control flow graph. This approach is generally efficient but restricted to rather shallow properties of programs such as aliasing relations [17,39], points–to information [47,51], or pointer range analysis [50].

**Hoare–style approaches:** first–order reasoning typically breaks down when it comes to prove properties of pointer–manipulating programs. The main reason is that it is impossible to express an invariant of all members of a data structure in first–order logic. The latter has to be extended therefore to support the definition of a reachability predicate [1,12,22,32,37,38]. However such deductive techniques usually involve user interaction, or otherwise only restricted properties such as dereferencing of nil pointers or aliasing effects can be analyzed.

**Separation logic** has been proposed as an extension to Hoare logic that permits local reasoning about linked structures, supporting features to support modular correctness proofs for pointer–manipulating programs [40,42]. It has been employed for termination proofs of heap–manipulating programs [4], for interprocedural shape

analysis [24], for handling abstract data types [7], and for verifying garbage collection algorithms [6]. However most of the work on separation logic focuses on verifying programs manually.

In summary, many of the characterizing features of our approach are already present in earlier papers: the restriction to singly–linked lists without data fields, the introduction of abstract entities which represent a potentially unbounded number of heap cells (called "summary nodes" in [13]); see e.g. [2,9,36], and the observation that, in this setting, the number of sharing points in heap structures is bounded by the number of program variables [9,36].

However none of these combines the strengths of our approach which supports concurrent programs with dynamic memory allocation and destructive updates such that arbitrary (cyclic) linked lists can be constructed, integrates both abstraction and model checking in a fully automated way, supports a linear–time logic in which both safety and liveness properties can be expressed, and which allows to use standard LTL model checkers.

In comparison, many of the existing approaches suffer from the poor programming environment, the exclusion of cyclic data structures, the requirement of user interaction, or the restriction to safety properties. Notable exceptions are [2], which also offers liveness properties but requires user–defined ranking functions, [20], which employs extended tableau–based techniques for model checking, and [49], which has a non–standard interpretation.

## 7 Conclusions and Future Work

We have presented a framework for the verification of concurrent pointer–manipulating programs with unbounded heap size and destructive updates. The correctness properties are specified using temporal pointer logic which is essentially pointer logic for expressing heap properties enriched with temporal operators. Instead of requiring dedicated algorithms, the TPL model checking problem is reduced to an LTL model checking problem that can be verified effectively with a broad variety of existing model checkers. The tradeoff is the restriction to list–like data structures as well as the limitation in expressiveness of the logic because object identities are not tracked between configurations.

Currently we are implementing our method to verify more realistic examples in the future. In particular we will extend the analysis of concurrent garbage collectors by defining a "hardest mutator", i.e., a general mutator program which is capable of simulating the behavior of any other mutator. This will enable us to establish the correctness of garbage collectors independent of the concrete mutator.

Furthermore due to the extensive use of concurrency, state space reduction and optimization techniques such as partial order reduction [21,23] will have to be employed and integrated in the implementation. We also plan to extend our framework with dynamic (unbounded) creation of threads. Another interesting aspect could be the combination of existing finite–state modeling languages like Promela [29] and pointer manipulation. Finally in the long run we have plans to increase the expressivity of the logic as well as to generalize our approach to richer data structures, for which new abstractions will be necessary. Moreover an automata–theoretic approach to defining a storeless semantics, as it is studied in [11] for a (concrete) semantics for pointer programs seems promising.

# References


[1] Amtoft, T., S. Bandhakavi and A. Banerjee, *A logic for information flow in object–oriented programs*, in: *POPL '06* (2006), pp. 91–102.

[2] Balaban, I., A. Pnueli and L. D. Zuck, *Shape analysis by predicate abstraction*, in: *VMCAI '05*, LNCS **3385** (2005), pp. 164–180.
URL http://springerlink.metapress.com/openurl.asp?genre=article&issn=0302-9743&volume=3385&spage=164

[3] Ball, T. and S. K. Rajamani, *The SLAM project: debugging system software via static analysis*, in: *POPL '02* (2002), pp. 1–3.

[4] Berdine, J., B. Cook, D. Distefano and P. W. O'Hearn, *Automatic termination proofs for programs with shape–shifting heaps*, in: *CAV '06*, LNCS **4144** (2006), pp. 386–400.

[5] Beyer, D., T. A. Henzinger and G. Théoduloz, *Lazy shape analysis*, in: *CAV '06*, LNCS **4144** (2006), pp. 532–546.

[6] Birkedal, L., N. Torp-Smith and J. C. Reynolds, *Local reasoning about a copying garbage collector*, in: *POPL '04* (2004), pp. 220–231.

[7] Bornat, R., C. Calcagno, P. O'Hearn and M. Parkinson, *Permission accounting in separation logic*, in: *POPL '05* (2005), pp. 259–270.

[8] Bouajjani, A., M. Bozga, P. Habermehl, R. Iosif, P. Moro and T. Vojnar, *Programs with lists are counter automata*, in: *CAV '06*, LNCS **4144** (2006), pp. 517–531.

[9] Bouajjani, A., P. Habermehl, P. Moro and T. Vojnar, *Verifying programs with dynamic 1–selector-linked list structures in regular model checking*, in: *TACAS '05*, LNCS 3440 **3440** (2005), pp. 13–29.

[10] Bouajjani, A., P. Habermehl, A. Rogalewicz and T. Vojnar, *Abstract regular tree model checking of complex dynamic data structures*, in: *SAS '06*, LNCS **4134** (2006), pp. 52–70.

[11] Bozga, M., R. Iosif and Y. Lakhnech, *Storeless semantics and alias logic*, ACM SIGPLAN Not. **38** (2003), pp. 55–65.

[12] Bozga, M., R. Iosif and Y. Lakhnech, *On logics of aliasing*, in: *SAS '04*, LNCS **3148** (2004), pp. 344–360.

[13] Chase, D. R., M. Wegman and F. K. Zadeck, *Analysis of pointers and structures*, in: *PLDI '90* (1990), pp. 296–310.

[14] Dams, D. and K. S. Namjoshi, *Shape analysis through predicate abstraction and model checking*, in: *VMCAI '03*, LNCS **2575** (2003), pp. 310–323.

[15] Das, S. and D. L. Dill, *Successive approximation of abstract transition relations*, in: *LICS '01* (2001), pp. 51–58.

[16] Das, S., D. L. Dill and S. Park, *Experience with predicate abstraction*, in: N. Halbwachs and D. Peled, editors, *CAV '99*, LNCS **1633** (1999), pp. 160–171.
URL http://link.springer.de/link/service/series/0558/bibs/1633/16330160.htm

[17] Deutsch, A., *Interprocedural may–alias analysis for pointers: beyond k–limiting*, in: *PLDI '94* (1994), pp. 230–241.

[18] Distefano, D., "On Model Checking the Dynamics of Object–Based Software: a Foundational Approach," Ph.D. thesis, Univ. of Twente (2003).

[19] Distefano, D., J.-P. Katoen and A. Rensink, *Who is pointing when to whom? – on the automated verification of linked list structures*, in: *FSTTCS '04*, LNCS **3328** (2004), pp. 250–262.

[20] Distefano, D., J.-P. Katoen and A. Rensink, *Safety and liveness in concurrent pointer programs*, in: *FMCO '06*, LNCS **4111** (2006), pp. 280–312.

[21] Flanagan, C. and P. Godefroid, *Dynamic partial–order reduction for model checking software*, in: *POPL '05* (2005), pp. 110–121.

[22] Fradet, P., R. Gaugne and D. L. Métayer, *Static detection of pointer errors: an axiomatisation and a checking algorithm*, in: *ESOP '96*, LNCS **1058** (1996), pp. 125–140.

[23] Godefroid, P., "Partial–Order Methods for the Verification of Concurrent Systems: An Approach to the State–Explosion Problem," LNCS **1032**, Springer–Verlag, 1996.

[24] Gotsman, A., J. Berdine and B. Cook, *Interprocedural shape analysis with separated heap abstractions*, in: *SAS '06*, LNCS **4134** (2006), pp. 240–260.

[25] Graf, S. and H. Saïdi, *Construction of abstract state graphs with PVS*, in: *CAV '97*, LNCS **1254** (1997), pp. 72–83.

[26] Hackett, B. and R. Rugina, *Region–based shape analysis with tracked locations*, in: *POPL '05* (2005), pp. 310–323.

[27] Heintze, N. and O. Tardieu, *Demand–driven pointer analysis*, ACM SIGPLAN Not. **36** (2001), pp. 24–34.

[28] Henzinger, T. A., R. Jhala, R. Majumdar and G. Sutre, *Software verification with BLAST*, in: *SPIN '03*, LNCS **2648** (2003), pp. 235–239.
URL http://link.springer.de/link/service/series/0558/bibs/2648/26480235.htm

[29] Holzmann, G., "The Spin Model Checker: Primer and Reference Manual," Addison–Wesley, 2003.

[30] Jeannet, B., A. Loginov, T. W. Reps and S. Sagiv, *A relational approach to interprocedural shape analysis*, in: *SAS '04*, LNCS **3148** (2004), pp. 246–264.

[31] Lee, O., H. Yang and K. Yi, *Automatic verification of pointer programs using grammar-based shape analysis*, in: *ESOP '05*, LNCS **3444** (2005), pp. 124–140.

[32] Lev-Ami, T., N. Immerman, T. W. Reps, S. Sagiv, S. Srivastava and G. Yorsh, *Simulating reachability using first–order logic with applications to verification of linked data structures*, in: *CADE '05*, LNCS **3632** (2005), pp. 99–115.
URL http://dx.doi.org/10.1007/11532231_8

[33] Lev-Ami, T., N. Immerman and S. Sagiv, *Abstraction for shape analysis with fast and precise transformers*, in: *CAV '06*, LNCS **4144** (2006), pp. 547–561.

[34] Lev-Ami, T. and S. Sagiv, *TVLA: A system for implementing static analyses*, in: *SAS '00*, LNCS **1824** (2000), pp. 280–302.

[35] Loginov, A., T. W. Reps and S. Sagiv, *Abstraction refinement via inductive learning*, in: *CAV '05*, LNCS **3576** (2005), pp. 519–533.

[36] Manevich, R., E. Yahav, G. Ramalingam and M. Sagiv, *Predicate abstraction and canonical abstraction for singly–linked lists*, in: *VMCAI '05*, LNCS **3385** (2005), pp. 181–198.

[37] Møller, A. and M. I. Schwartzbach, *The pointer assertion logic engine*, in: *PLDI '01* (2001), pp. 221–231.

[38] Nelson, G., *Verifying reachability invariants of linked structures*, in: *POPL '83* (1983), pp. 38–47.

[39] Nystrom, E. M., H.-S. Kim and W. mei W. Hwu, *Bottom–up and top–down context–sensitive summary–based pointer analysis*, in: *SAS '04*, LNCS **3148** (2004), pp. 165–180.

[40] O'Hearn, P. W., H. Yang and J. C. Reynolds, *Separation and information hiding*, in: *POPL '04* (2004), pp. 268–280.

[41] Podelski, A. and T. Wies, *Boolean heaps*, in: *SAS '05*, LNCS **3672** (2005), pp. 268–283.

[42] Reynolds, J. C., *Separation logic: A logic for shared mutable data structures*, in: *LICS '02* (2002), pp. 55–74.

[43] Rinetzky, N., J. Bauer, T. Reps, M. Sagiv and R. Wilhelm, *A semantics for procedure local heaps and its abstractions*, in: *POPL '05* (2005), pp. 296–309.

[44] Rinetzky, N., M. Sagiv and E. Yahav, *Interprocedural shape analysis for cutpoint–free programs*, in: *SAS '05*, LNCS **3672** (2005), pp. 284–302.

[45] Sagiv, M., T. Reps and R. Wilhelm, *Solving shape–analysis problems in languages with destructive updating*, ACM Trans. Program. Lang. Syst. **20** (1998), pp. 1–50.

[46] Sagiv, M., T. Reps and R. Wilhelm, *Parametric shape analysis via 3–valued logic*, ACM Trans. Program. Lang. Syst. **24** (2002), pp. 217–298.

[47] Whaley, J. and M. S. Lam, *An efficient inclusion–based points–to analysis for strictly–typed languages*, in: *SAS '02*, LNCS **2477** (2002), pp. 180–195.

[48] Yahav, E. and G. Ramalingam, *Verifying safety properties using separation and heterogeneous abstractions*, in: *PLDI '04* (2004), pp. 25–34.

[49] Yahav, E., T. Reps, M. Sagiv and R. Wilhelm, *Verifying temporal heap properties specified via evolution logic*, in: *ESOP '03*, LNCS **2618** (2003), pp. 204–222.

[50] Yong, S. H. and S. Horwitz, *Pointer–range analysis*, in: *SAS '04*, LNCS **3148** (2004), pp. 133–148.
URL
http://springerlink.metapress.com/openurl.asp?genre=article&issn=0302-9743&volume=3148&spage=133

[51] Zhu, J. and S. Calman, *Symbolic pointer analysis revisited*, in: *PLDI '04* (2004), pp. 145–157.