ISSN 1470-5559

COSMICAH 2005: workshop on verification of COncurrent Systems with dynaMIC Allocated Heaps (a Satellite event of ICALP 2005) - Informal Proceedings

Editors: Dino Distefano, Radu Iosif & Peter O'Hearn

RR-05-04

July 10 2005,
Lisboa, Portugal

Department of Computer Science

Program

9.00 Welcome

9.10 INVITED TALK:
Transition Invariants, Transition Predicate Abstraction, Counterexample-Guided Abstraction Refinement
A. Podelski

10.15 Coffee break

REGULAR PAPERS

10.45 Heap-abstractions for an object-oriented calculus with thread classes
E. Abraham, A. Gruner, M. Steffen

11.15 Symbolic Execution with Separation Logic
J. Berdine, C. Calcagno, P. O'Hearn

11.45 Verifying Red-Black Trees
P. Baldan, A. Corradini, J. Esparza, T. Heindel, B. Konig, V. Kozioura

12.15 Lunch break

13.45 Local Reasoning for Termination
A. Podelski and I. Schaefer

PRESENTATIONS

14.15 Read-only permissions and abstract predicate
M. Parkinson

14.45 Local Reasoning about Tree Update
C. Calcagno

15.15 Coffee Break

15.45 Boolean Heaps
T. Wies

16.15 FIVE MINUTES MADNESS Session

17.00 (the latest) Closing.

Preface

This volume contains the proceedings of the First workshop on the verification of COncurrent Systems with dynaMIC Allocated Heaps (COSMICAH). COSMICAH took place in Lisbon, Portugal, July 10, 2005, as a satellite workshop of the International Colloquium on Automata, Languages and Programming (ICALP). The workshop brings together researchers with different backgrounds in formal methods, that share interest in the verification of programs which combine advanced language features such as: concurrency, dynamic memory, recursion, etc. The problems addressed by the participants in the workshop are of crucial importance in the (well-established by now) domain of software verification.

Topics covered by COSMICAH include logics for the specification of properties related to the dynamic memory of programs, object calculi and other semantic models for software, verification algorithms such as satisfiability solving, abstract interpretation, theorem proving and model checking.

To provide an effective basis for discussion, the emphasis of the workshop was on the quality of the presentations. Beside those of the reviewed papers, a keynote invited talk was given by Andreas Podelski. Moreover, we had the pleasure to host three presentations — given by young promising researchers — on hot topics in the field. The character of these proceedings is informal, thus inclusion of a paper does not preclude further submission to a larger symposium. We would like to thank the authors of submitted papers and the other speakers.

To carry out the task of reviewing the submitted papers, we were fortunate to have a highly qualified Program Committee from different research areas. Each submission received comments from two reviewers. Special thanks to all members of the Program Committee. The help of the ICALP organising committee in dealing with the technical details of the organisation was invaluable.

We hope to see you all next year to a new edition of COSMICAH!

Dino Distefano, Radu Iosif, and Peter O'Hearn

Program Committee

Cristiano Calcagno (Imperial College, London)
Dino Distefano (Queen Mary, Univ. of London, co-chair)
Peter Habermehl (Liafa, Paris 7 University)
Peter O'Hearn (Queen Mary, Univ. of London, co-chair)
Radu Iosif (Verimag, Grenoble, co-chair)
Yassine Lakhnech (Verimag, Grenoble)
Arend Rensink (University of Twente, Enschede)
Robby (Kansas State University, Manhattan KS)
Eran Yahav (IBM Research, New York)

Sponsoring Institution

COSMICAH is an official Appsem II workshop.

Table of Contents

Verifying Red-Black Trees
Paolo Baldan, Andrea Corradini, Javier Esparza, Tobias Heindel,Barbara König, and Vitali Kozioura 1

Local Reasoning for Termination
Andreas Podelski and Ina Schaefer 16

Symbolic Execution with Separation Logic
Josh Berdine, Cristiano Calcagno,and Peter W. O'Hearn 31

Heap-Abstraction for an Object-Oriented Calculus with Thread Classes
Erika Ábrahám, Andreas Grüner, and Martin Steffen 47

Verifying Red-Black Trees*

Paolo Baldan[1], Andrea Corradini[2], Javier Esparza[3], Tobias Heindel[3], Barbara König[3], and Vitali Kozioura[3]

[1] Dipartimento di Informatica, Università Ca' Foscari di Venezia, Italy
[2] Dipartimento di Informatica, Università di Pisa, Italy
[3] Institut für Formale Methoden der Informatik, Universität Stuttgart, Germany

baldan@dsi.unive.it andrea@di.unipi.it
{esparza,heindets,koenigba,koziouvi}@fmi.uni-stuttgart.de

Abstract. We show how to verify the correctness of insertion of elements into red-black trees—a form of balanced search trees—using analysis techniques developed for graph rewriting. We first model red-black trees and operations on them using hypergraph rewriting. Then we use the tool Augur, which computes approximated unfoldings, in order to show that insertion preserves the property that there are no two consecutive red nodes in a tree, a requirement for red-black trees. Furthermore, we prove that the tree remains balanced by exploiting a type system that can be obtained as an instance of a general framework.

1 Introduction

In order to verify programs written in languages with dynamic memory allocation, such as C, it is important to find suitable abstractions for the dynamically evolving pointer structures on the heap. The same problem arises for object-oriented languages, for instance Java. Despite existing techniques such as alias and points-to analysis [20, 24] and shape analysis [19], this is still a major open problem. This paper proposes to use verification techniques based on graph rewriting. The basic idea is to represent the state of the heap by a graph and dynamic transformations of the pointer structure by graph rewriting rules. Compared to the approaches to shape analysis which represent these structures as models of a 3-valued logic we follow a more direct approach where pointer structures are represented as graphs, and graph morphisms can be used as a convenient abstraction mechanism. This allows us to exploit partial order semantics already developed for graph rewriting, as well as its close relation to Petri nets, which we use as abstractions (over-approximations) of the behavior of graph rewriting systems.

We demonstrate the effectiveness of this approach by modeling the insertion of elements into red-black trees and verifying (partial) correctness of the insertion operation. Red-black trees are binary search trees whose nodes are colored either

* Research partially supported by DFG project SANDS and EC RTN 2-2001-00346 SegraVis.

black or red. Only inner nodes can be red, and the following two properties are satisfied: no red node has a red child, and the "black depth" is the same for all leaves. In order to re-establish these properties after a new element is inserted, it is necessary to perform some local transformations on the tree (called *rotations*), which have the effect of rebalancing it. After modeling rotations as graph rewriting rules, we use two different techniques to show that the two properties of red-black trees mentioned above still hold after an insertion. The first property is shown by automatically abstracting the graph rewriting system into a Petri net [2, 4] by means of the AUGUR tool. The second property is proved by resorting to a type-theoretical framework for graph rewriting, proposed in [9].

The rest of the paper is structured as follows. In Section 2 we introduce red-black trees and their representation as hypergraphs. In Section 3 we model insertion into red-black trees using graph rewriting. In Section 4 we show how to verify that insertion preserves the structural properties of red-black trees. Finally, in Section 5, we draw some conclusions.

2 Red-Black Trees

Red-black trees are a form of balanced search trees which can be easily implemented (see [11, 5]). They can also be seen as a variant of $(2, 4)$-trees.

Definition 1 (Red-black tree). *A red-black tree is a finite binary tree whose inner nodes are associated with keys. Keys are elements of a totally ordered set. A node can either be red or black. A red-black tree satisfies the following conditions:*

(S) *The tree is sorted, i.e., for every node v the maximal key in the left subtree is smaller than the key of v, and the minimal key in the right subtree is equal to or larger than the key of v.*
(RL) *The root and the leaves are black.*
(D) *All leaves have the same black depth, i.e., the number of black nodes on the path from the root is the same for all leaves.*
(R) *No path from the root to a leaf contains two consecutive red nodes.*

Due to these conditions the longest path from the root to a leaf is at most twice as long as the shortest one. The height of a red-black tree with n inner nodes is therefore $O(\log(n + 1))$, and thus we say that the tree is balanced.

Since we will model insertion into red-black trees by hypergraph rewriting, in the paper we always depict red-black trees as hypergraphs.

In the following, given a set A, let A^* denote the set of finite sequences of elements of A and for $s \in A^*$, let $|s|$ denote its length.

Definition 2 (Hypergraph). *Let Λ be a fixed set of edge* labels*, where every label $l \in \Lambda$ is associated with an* arity *$ar(l) \in \mathbb{N}$.*

*A (Λ-)*hypergraph *(or simply* graph*) is a tuple $G = (V_G, E_G, c_G, l_G)$, where V_G is a set of* vertices *and E_G is a set of* hyperedges*. Each hyperedge is attached to a sequence of vertices, as expressed by the* connection function *$c_G \colon E_G \to V_G^*$,*

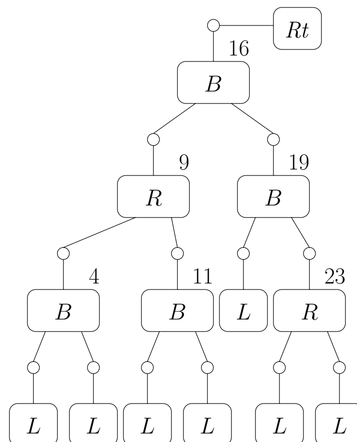

Fig. 1. An example of a red-black tree.

and it is labeled with an element of Λ via the labeling function $l_G : E_G \to \Lambda$. *For any hyperedge $e \in E_G$ it must hold that $ar(l_G(e)) = |c_G(e)|$, i.e., the number of nodes an hyperedge is attached to is determined by the arity of its label.*

Hypergraph morphisms $\varphi : G \to G'$ are defined, as usual, as structure preserving mappings (see also [18]).

A red-black tree is represented as a hypergraph where hyperedges correspond to the nodes of the tree. Inner nodes are represented by hyperedges of arity 3, i.e., they are connected to exactly three vertices, where the parent and the left and right children can be attached. They are labeled by either R or B depending on whether the node is red or black. Leaves are represented by unary hyperedges labeled L. Furthermore there is, for technical convenience, a single unary hyperedge labeled Rt, indicating the root node. Figure 1 depicts a red-black tree, where the keys are written next to the hyperedges. Note that, by definition of hypergraph, each hyperedge is connected to an *ordered* sequence of vertices. In our pictures, vertices are always arranged in such a way that the vertex above a hyperedge is its first vertex, whereas the remaining vertices are ordered counter-clockwise.

3 Insertion into Red-Black Trees using Graph Rewriting

We introduce now the concepts of graph rewriting rule and rewriting step, which will be used to model the insertion of a new node into a red-black tree.

Definition 3 (Graph rewriting rule). *A* graph rewriting rule *r is a tuple (L, R, α) where L and R are hypergraphs, called the* left-hand side *and* right-hand side *of the rule, while $\alpha : V_L \to V_R$ is an injective function.*

Intuitively, to apply a rule $r = (L, R, \alpha)$ to a hypergraph G one must find an occurrence of the left-hand side L in G, i.e., a hypergraph morphism $\varphi : L \to G$. The application of the rule first removes from G the image of the hyperedges of L, and then extends the resulting hypergraph by adding the new vertices in R (i.e., the vertices in $V_R - \alpha(V_L)$) and all the hyperedges of R, yielding a new hypergraph H. In this case we write $G \Rightarrow^r H$. Observe that, unlike hyperedges, vertices are never deleted: the vertices of G are not affected by the rewriting step. We refer to [2] for a discussion of this restriction with respect to more general definitions of graph rewriting. Notice, anyway, that the deletion of a vertex can be simulated in our framework by leaving it isolated in the resulting graph.

The insertion of a new node into a red-black tree is described by the hypergraph rewriting rules shown in Fig. 2 and Fig. 3. For the corresponding pseudocode, see for instance [11]. An interesting question, that we leave as a topic of future research, is whether and how graph rewriting rules can be synthesized automatically from (pseudo-)code.

In the following the mapping α of a rule is represented by numbering the nodes in the left-hand and right-hand sides: α maps a node in the left-hand-side to the node of the right-hand side with the same number. Furthermore keys are denoted by the letters y, z, u, v.

Rule [add-leaf] describes how a leaf is replaced by a new inner node labeled M and two leaves. The label M stands for “marker” and denotes a red node during the insertion phase. Rule [add-leaf] also consumes a “token”, the 0-ary hyperedge *add*, that will be generated again when the insertion is completed: this mechanism prevents the concurrent insertion of nodes. We assume that the insertion of the new key y starts from the appropriate leaf, whose position must have been determined by a previous search on the tree. Although this is out of our focus, it is worth observing that this search could be realized by means of graph rewriting rules acting on attributed graphs [10].

The remaining rules describe the local transformations needed to ensure that the tree is converted into a red-black tree. If the marker has a black parent, it is converted into a red hyperedge and insertion terminates (rule [marker-black], this rule has two symmetric variants). If the marker is the root (rule [marker-root]), it is replaced by a black hyperedge; in this case the black depth of the tree increases by one. If the marker has a red parent, we distinguish several cases (notice that in this case the marker's grandparent (if any) must be black, because otherwise Condition (R) would be violated):

- If the red parent of the marker has a red sibling, we perform a flip and move the marker upwards (rule [flip], four variants). In this case the algorithm continues.
- If the red parent of the marker has a black sibling, and this sibling is not a leaf, we apply either rule [rotation] or rule [double-rotation]. Rule [rotation] is applied if the marker and its red parent are either both left children or both right children. In the two remaining cases rule [double-rotation] is applied. In all cases the algorithm terminates.

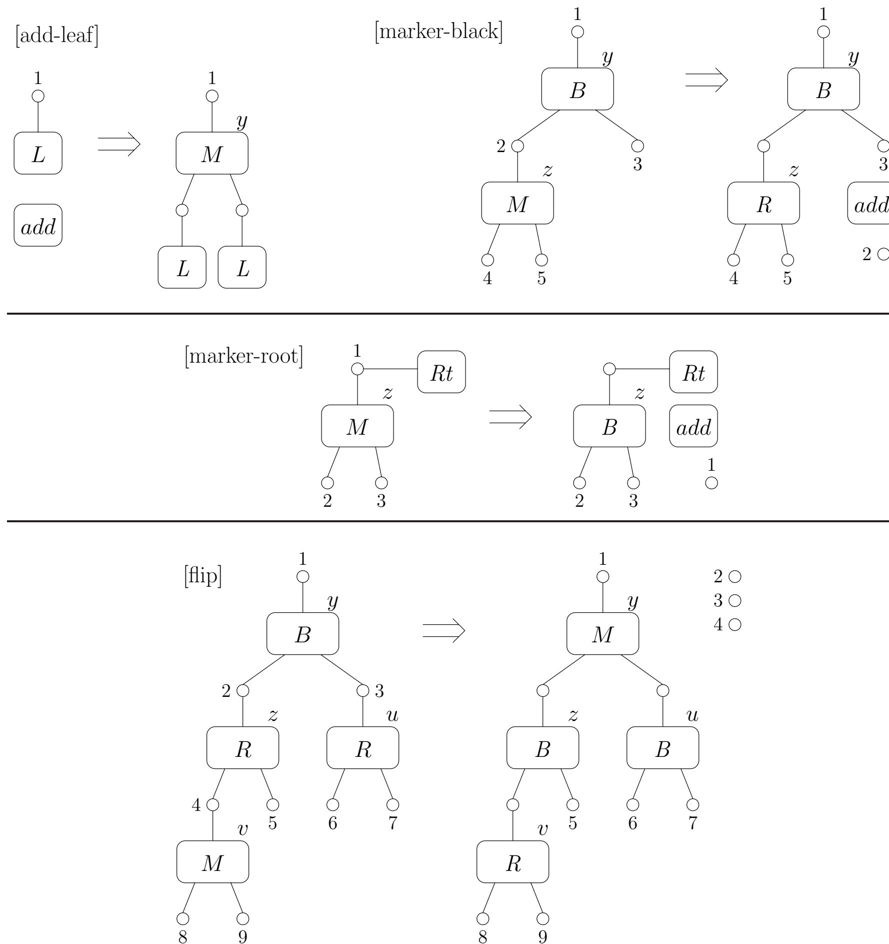

Fig. 2. Graph rewriting rules (insertion of an element into a red-black tree), part I.

– If the red parent of the marker has a black sibling, but this sibling is a leaf, we proceed similarly to the previous case. There are four more rules, obtained from those of Fig. 3 by replacing the node with key u by a leaf.

One can see fairly easily that all the transformations expressed by the above rules preserve the sortedness Condition (S) in Definition 1. Moreover, for any given finite tree, the insertion procedure started by rule [add-leaf] surely terminates, generating again the token *add*. The formal verification of these two properties goes beyond the goals of this paper: we shall only sketch in the conclusion how this could be done by exploiting the available theory of confluence and termination of graph rewriting systems.

Note that modeling insertion into red-black trees using graph rewriting rules is very natural. Similar diagrams can be found in most text books introducing

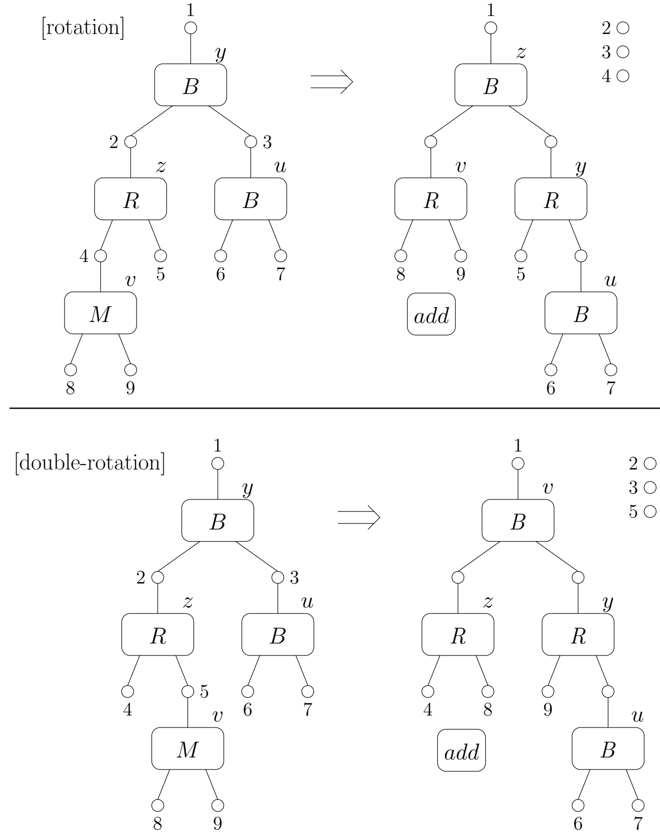

Fig. 3. Graph rewriting rules (insertion of an element into a red-black tree), part II.

red-black trees. Usually no marker is used, a red node takes its place instead. However, this would lead to "inconsistent" intermediate states, produced during the insertion procedure, which *do* contain two consecutive red hyperedges, violating Condition (R). We avoid this by using a specific marker, which is furthermore useful for indicating the position in the tree where operations have to be performed.

4 Verifying Red-Black Trees

In the following we describe two static analysis techniques developed for the verification of graph transformation systems: approximated unfolding and type systems. Approximated unfolding is a fully automatic technique, based on a partial order semantics of graph transformation systems. Here it is used to show that no tree generated by the rewriting rules for insertion has two consecutive red nodes (Condition (R)). The property that red-black trees remain balanced (Con-

dition (D)) is checked using a suitable type system, which is a simple instance of a general framework [9]. We assume that the preservation of Conditions (S) and (RL) has already been proved, as well as the fact that the result of the insertion procedure is again a tree.

As the rest of the paper concentrates only on structural properties of red-black trees, we neglect keys in the following.

4.1 Approximated Unfolding

Approximated unfolding was proposed in [2, 4] for the verification of infinite state systems, modelled as graph transformation systems. It is based on the unfolding construction which, applied to a graph transformation system, produces a static structure fully describing the concurrent behavior of the system, including all possible rewriting steps and their mutual dependencies, as well as all reachable states [17, 3].

The unfolding is infinite for any non-trivial graph transformation system. The mentioned papers propose an algorithm for constructing finite structures which can be seen as over-approximations of the full unfolding, where interesting classes of properties of the original system can be studied and verified. The structures used for approximation are so-called *Petri graphs*, consisting of Petri nets the places of which are hyperedges.

The outcome of the algorithm is determined by the chosen level of accuracy: essentially one can require the approximation to be exact up to a certain causal depth k, thus obtaining the so-called *k-covering* $\mathcal{C}^k(\mathcal{G})$ of the unfolding of $\mathcal{G}$.

The covering $\mathcal{C}^k(\mathcal{G})$ over-approximates the behavior of $\mathcal{G}$ in the sense that every computation in $\mathcal{G}$ is mapped to a valid computation in $\mathcal{C}^k(\mathcal{G})$ and every hypergraph reachable from the start hypergraph can be mapped homomorphically to (the graphical component of) $\mathcal{C}^k(\mathcal{G})$ (and its image is reachable in the Petri graph). Therefore, given a property over hypergraphs reflected by hypergraph morphisms, if it holds for all hypergraphs reachable in the covering $\mathcal{C}^k(\mathcal{G})$ then it also holds for all reachable hypergraphs in $\mathcal{G}$. Important properties of this kind are the non-existence and non-adjacency of edges with specific labels, the absence of certain paths (for checking security properties) or cycles (for checking deadlock-freedom). These structural properties can be specified using regular expressions or by a monadic second-order logic on graphs that can be combined with a temporal logic [4].

The technique described above has been implemented as part of the Augur tool.[4] It takes as input a graph transformation system encoded in GTXL (Graph Transformation eXchange Language, an XML standard for graph transformation systems) and outputs the Petri graph in GXL (Graph eXchange Language). Next, several tools integrated with Augur can be used for verifying the desired properties over the resulting Petri graph.

In order to show with Augur that insertion in a red-black tree does not violate Condition (R), we provide as input to the tool a modified version of the

[4] See `http://www.fmi.uni-stuttgart.de/szs/tools/augur/`.

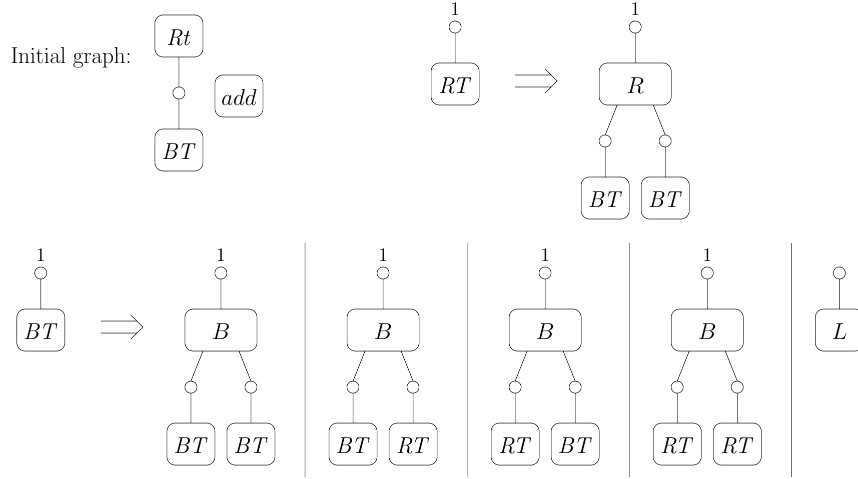

Fig. 4. A context-free grammar for generating red-black trees.

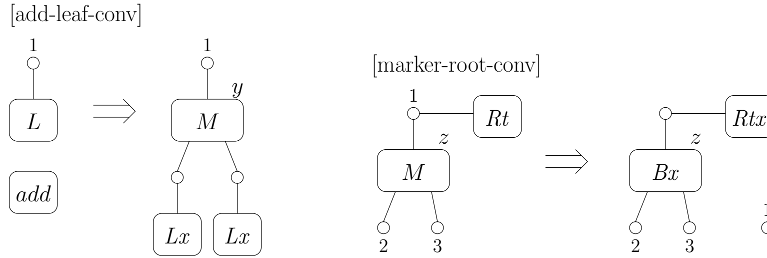

Fig. 5. Rules of the converted system, part I.

rules shown in Figs. 2 and 3, as well as rules for generating all possible red-black trees. The context-free rules for generating trees are shown in Fig. 4, together with the initial graph: they use the non-terminals BT and RT, and generate all finite trees satisfying Conditions (RL) and (R), but possibly not Condition (D) (i.e., they are not balanced). Moreover, the rules modeling insertion are obtained from those of the previous section as described next.

First, since every possible red-black tree is generated by the rules of Fig. 4, it is sufficient to show that Condition (R) holds again after a single insertion; thus in the modified rules, the token *add* is never generated again. Second, in order to speed-up the verification, it is convenient to "freeze" the part of the tree traversed during insertion. This is obtained by changing all labels Rt, B, R and L appearing in the right-hand side of rules to labels Rtx, Bx, Rx and Lx, respectively, which do not appear in any rule's left-hand side (see Fig. 5). This transformation is safe, because the hyperedges with x-marked labels do not interfere with the current insertion, and no further insertion is possible by the previous point.

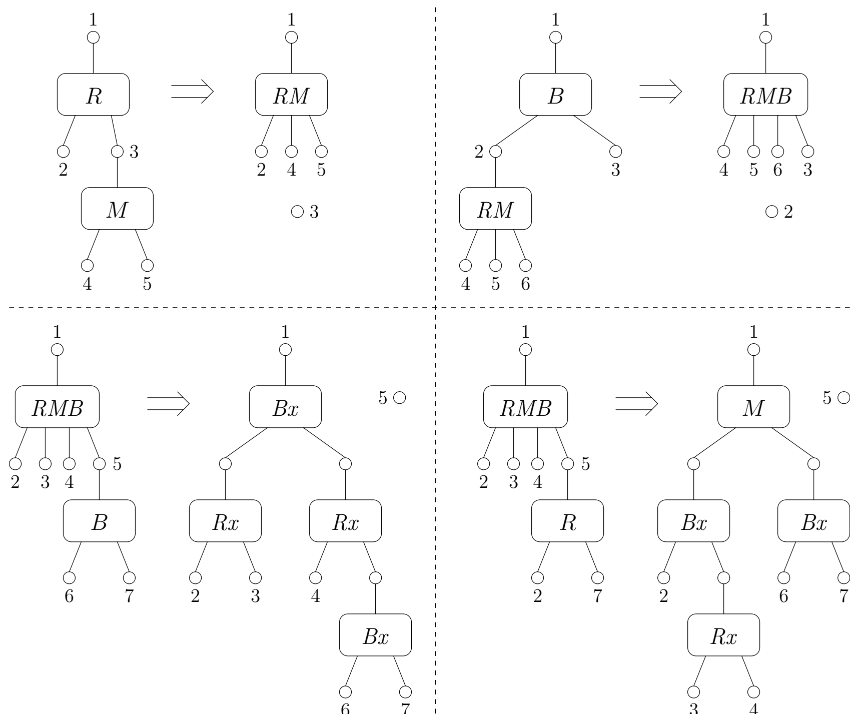

Fig. 6. Rules of the converted system, part II.

The third modification is necessary because the current implementation of the approximated unfolding suffers from the restriction that a rule cannot have two hyperedges with the same label in the left-hand side, but rules [flip], [rotation] and [double-rotation] do not satisfy this restriction. Therefore the offending rules are converted into an equivalent set of rules which use some new labels and satisfy this restriction. The way the new rules work can be grasped from Fig. 6. If the first three rules can be applied in sequence, then we identified an occurrence of the left-hand side of [double-rotation], and therefore the corresponding right-hand side is generated (modified according to the previous two points). If instead after the first two rules the left-hand side of the fourth rule is found, then we generate the right-hand side of a [flip]. It can be shown that the converted rules are equivalent to the original ones, in the sense that if G and G' are graphs containing only labels of the original graph rewriting system, then G can be rewritten to G' in the original system if and only if G can be rewritten to G' in the converted system, possibly in more steps. Furthermore, all hyperedges labeled by a label introduced in the converted system will eventually be deleted.

Applying Augur to the graph rewriting system just described and asking for the 0-th approximation we get a Petri graph $\mathcal{C}^0$ with 125 hyperedges, 72 vertices and 46 transitions, which is too large to be depicted here. In order to show

that the property under consideration holds, we want to check that no reachable graph contains a path corresponding to the regular expression $(R+Rx)(R+Rx)$. The tools integrated into AUGUR can convert this regular expression into a set of markings such that a path of this kind exists in the approximation if and only if the corresponding markings are reachable in $\mathcal{C}^0$. However, in this case the set of markings is empty, meaning that the hypergraph underlying the Petri graph does not contain two consecutive red edges. In other words, using only the structural properties of the covering $\mathcal{C}^0$ (without taking into account its behavior) we can infer the desired property.

4.2 A Type System

In [9] a general framework for typing graph rewriting systems has been presented which will be instantiated in the following in order to analyze red-black trees. Type systems of this kind can be used to check structural invariants and are related to type systems for process calculi [12].

Some intuition. Loosely speaking, a type system for a graph rewriting system is a mapping that associates to a graph G another graph T, the (graph) type of G. We say that it satisfies the *subject reduction* property if whenever G is rewritten to G' and G has type T, then G' also has type T. In order to prove that insertion preserves Condition (D) (all leaves have the same black depth), it suffices to design a type system and a condition (P) over graph types such that:

(1) the type system has the subject reduction property with respect to the rules for insertion;
(2) a graph satisfies Condition (D) if and only if its type satisfies Condition (P).

To see why, let G be any tree satisfying Condition (D), and let G' be the result of performing an insertion into G. By (1), G and G' can be assigned the same type. By (2), this type satisfies Condition (P) and, by (2) again, G' satisfies Condition (D).

Intuitively, our type system assigns to a graph G the graph T obtained by (a) removing all red hyperedges, directly linking their parents to their children, and (b) merging all black hyperedges having the same distance from the root. It is easy to see that G satisfies Condition (D) if and only if no leaf of G is merged to an inner hyperedge of T. This is Condition (P).

The technical setting. In the following we consider graphs G with a distinguished sequence of *external vertices* $\chi_G \in V_G^*$, possibly with repetition. Graphically, we identify the i-th vertex in the sequence by writing the number i close to the corresponding node. The length of χ_G is called the *arity* of G. Rewriting rules of the form (L, R, α) can now be seen as pairs of graphs with the same arity, where χ_L is an arbitrary but fixed linearisation of V_L, and $\alpha(v) = v'$ if and only if v, v' appear in the same position of χ_L, χ_R. In the following, all operations and morphisms are expected to preserve external vertices, i.e., for a morphism

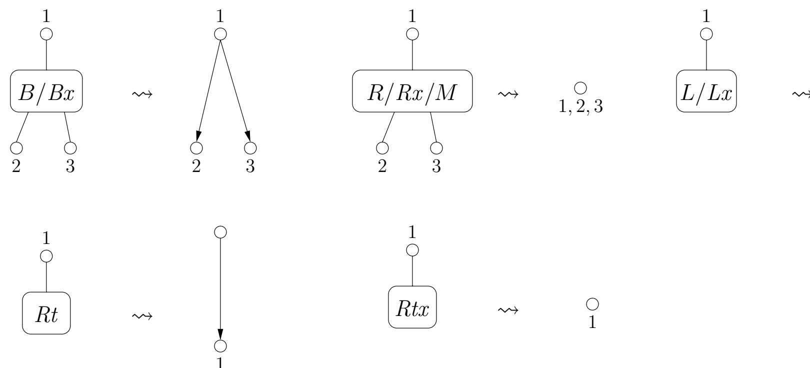

Fig. 7. Local step.

$\varphi: G \to G'$ we demand $\varphi(\chi_G) = \chi_{G'}$. Technically, a type system associates to G not only the graph T, but also all the graphs T' such that there is a graph morphism $T \to T'$ (the type T can be seen as a subtype of each such T'). All these graphs are the *graph types* of G, and T is the *strongest graph type*. We consider type systems in which the strongest graph type is obtained by first applying a local transformation which replaces every hyperedge e by a graph having the same arity as e. In a second phase a global closure operator is applied which usually "folds" the graph obtained after the first step. In [9] it is shown that under some mild conditions the subject reduction property holds whenever we can show the following local property for every rewriting rule (L, R, α):

> **(Local subject reduction)** Let T_L, T_R be the strongest graph types for L and R. Then there is a morphism $\varphi: T_R \to T_L$.

The type system. We describe the local and global step of our type system. They correspond to the algorithmic steps (a) and (b) described above. We consider here a graph rewriting system modeling a single insertion into a tree, consisting basically of the rules of Fig. 2 and Fig. 3, where in the right-hand sides the token *add* is removed and labels are of the variant marked by x (see the first two modifications described in Section 4.1).

Local step: We replace every hyperedge modeling a black node by two binary edges and every leaf by a unary edge indicated by a black rectangle (see Fig. 7). Furthermore markers and red hyperedges are removed and all their vertices are collapsed (this corresponds to step (a) above). A hyperedge labeled *Rt* indicating the root is typed with a binary edge in order to have a black node "in reserve" whenever the black depth of a tree grows.[5]

[5] Observe that the type system makes a distinction between *Rt* and *Rtx* since in the case of *Rt* an extra black edge is inserted, which is not done in the case of *Rtx*. This makes it possible to establish the subject reduction property for rule [marker-root].

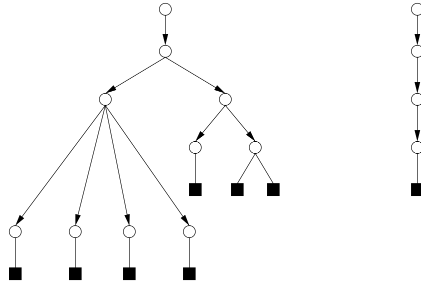

Fig. 8. Construction of a graph type (after the local/global step).

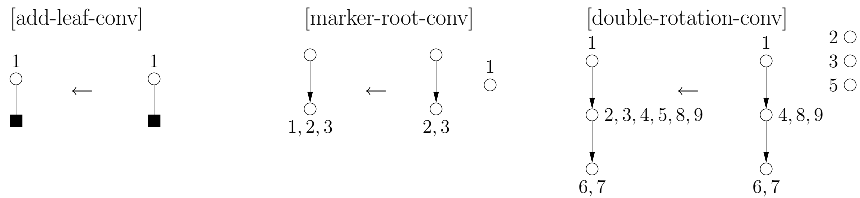

Fig. 9. Checking the local subject reduction property.

Global step/Closure: In the global step we collapse all branching black paths into one (step (b) above) as follows: Whenever there are two binary edges with the same source vertex or two unary edges with the same vertex, they are merged. This process may have to be repeated.

Alternatively this closure operation can also be characterised by means of a universal property.

If we apply the process described above to the red-black tree E in Fig. 1 we obtain the graph type T_E depicted on the right-hand side of Fig. 8 (the intermediate graph obtained after the local step is shown on the left-hand side).

Proving that insertions preserve Condition (D). According to the scheme shown at the beginning of the section, we have to prove that (1) the type system has the subject reduction property and (2) find a condition (P) such that Condition (D) holds for a graph iff Condition (P) holds for its (strongest) type.

For (1), it is straightforward to show that the components of the type framework, especially the operators used in the local and global step, satisfy the conditions identified in [9], and thus it suffices to prove the local subject reduction property. This is quite easy for most of the rules, we only show the property for the rules [add-leaf], [marker-root] and [double-rotation] (see Fig. 9).

As for (2), recall that (P) should intuitively be: no leaf of a graph is merged with an inner node in the graph type. The next proposition formalizes this fact.

Proposition 1. *Let E be a tree satisfying all conditions of a red-black tree with the possible exception of Condition (D). Furthermore let T_E be its strongest type.*

Then all leaves in E have the same black depth if and only if T_E satisfies the following condition:

> *(P) No unary edge (representing a leaf) is attached to a vertex which is also the source vertex of a binary edge (representing a black edge).*

Furthermore (P) satisfies the conditions specified in [9], specifically it is reflected by morphisms.

Hence we have shown that only balanced red-black trees are reachable from a balanced red-black tree by the rewriting rules in Figs. 2 and 3.

5 Conclusion and Related Work

We have shown how to model insertion into red-black trees using graph transformations and we have demonstrated how analysis and verification techniques based on graph transformation can be successfully used to verify the (partial) correctness of insertion. The first technique (approximated unfolding) is fully automatic and especially well-suited for showing that no reachable graph contains certain forbidden graph patterns. Other types of invariants can be more conveniently checked by using the second technique, a type system which is an instance of a general framework.

More generally, we are convinced that a single analysis method can not solve all problems, and thus a mix of several techniques is a promising method for the verification of pointer structures. For example, there are other relevant properties related to insertion into red-black trees that we did not address formally (as this was beyond the goal of the paper), but that we could handle using other available techniques. For example, it is quite obvious that termination of insertion can be proved easily by defining a suitable reduction ordering. More interestingly, let us sketch how the available theory of confluence for graph rewriting systems could be used to prove that insertion into a red-black tree preserves sortedness (Condition (S)).

Let us consider the system consisting of the rules of Figs. 2 and 3; since keys are relevant for this discussion, we assume that they are represented as unary hyperedges connected to the B, R or M hyperedge through a fourth node. The technique is based on the well-known fact that a binary tree is sorted (i.e., it satisfies Condition (S)) if and only if the in-order traversal returns its keys in sorted order. The in-order traversal can be modeled by graph rewriting rules that, starting from the root, destroy the tree while collecting all the keys in a linked list. Then the preservation of sortedness can be reduced to the proof that the system containing the rules for insertion and for in-order traversal is confluent: together with termination, intuitively this means that at whatever stage we stop the insertion, the in-order traversal returns the keys in the same order, and thus the tree remains sorted if it was so at the beginning. Pragmatically, confluence can be proved by resorting to a *critical pair lemma* for graph rewriting [13], and automated support for critical pair analysis is provided, e.g., by the AGG tool.[6]

[6] See `http://tfs.cs.tu-berlin.de/agg/critical_pairs.html`.

Research concerned with the verification of graph transformation systems is fairly recent. While some research groups [22, 6] pursue the idea of translating graph transformation systems into the input language of a model checker, others attempt to develop new specialized methods for graph rewriting. Work from our side goes in this latter direction, as well as [15, 14, 16]. Most of the work so far is concerned with verifying finite-state systems, whereas we have shown in this paper how to analyze an infinite-state system where elements can be inserted into red-black trees of arbitrary size.

In [23] red-black trees are checked using the structural analysis technique implemented in Alloy. Originally written in Java, the program is translated to Alloy's input language and is then analyzed (by a further translation to SAT). This requires bounds on the number of generated objects, on the maximum depth of the call stack and on the number of times a loop can be executed. In [21], the Moped model-checker is used, which allows to remove the last two bounds, but not the first. As a consequence, both techniques will find bugs if they appear for trees with a few nodes (around 5-7 in [23, 21]), but are not able to completely verify red-black trees of arbitrary size as we have done in this paper.

In [8] and [1] shape types and shapes are introduced as certain graph languages. Both papers propose algorithms that analyze each rule and check whether (and how) it may change the shape of a graph. In order to describe shapes the former uses context-free grammars whereas the latter uses more expressive graph reduction systems, that are able to express properties such as balancedness. In principle this technique could be used to show invariants of red-black trees, but the choice of graph reduction systems for shapes is non-trivial.

Furthermore insertion into red-black trees has been analyzed using the pointer assertion logic PALE and the tool MONA [7].

References

1. Adam Bakewell, Detlef Plump, and Colin Runciman. Checking the shape safety of pointer manipulations. In Rudolf Berghammer, Bernhard Möller, and Georg Struth, editors, *Proc. of RelMiCS '03*, volume 3051 of *LNCS*, pages 48–61. Springer, 2003.
2. Paolo Baldan, Andrea Corradini, and Barbara König. A static analysis technique for graph transformation systems. In *Proc. of CONCUR '01*, volume 2154 of *LNCS*, pages 381–395. Springer, 2001.
3. Paolo Baldan, Andrea Corradini, and Ugo Montanari. Unfolding and Event Structure Semantics for Graph Grammars. In W. Thomas, editor, *Proc. of FoSSaCS '99*, volume 1578 of *LNCS*, pages 73–89. Springer, 1999.
4. Paolo Baldan and Barbara König. Approximating the behaviour of graph transformation systems. In *Proc. of ICGT'02*, volume 2505 of *LNCS*, pages 14–29. Springer, 2002.
5. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* MIT Press, 2001. Second Edition.
6. Fernando Luís Dotti, Luciana Foss, Leila Ribeiro, and Osmar Marchi Santos. Verification of distributed object-based systems. In *Proc. of FMOODS '03*, volume 2884 of *LNCS*, pages 261–275. Springer, 2003.

7. Jacob Elgaard, Anders Møller, and Michael I. Schwartzbach. Compile-time debugging of C programs working on trees. In *Proc. of ESOP '00*, pages 119–134. Springer-Verlag, 2000. LNCS 1782.
8. Pascal Fradet and Daniel Le Métayer. Shape types. In *Proc. of POPL '97*, pages 27–39. ACM, 1997.
9. Barbara König. A general framework for types in graph rewriting. *Acta Informatica*, to appear.
10. Michael Löwe, Martin Korff, and Annika Wagner. An Algebraic Framework for the Transformation of Attributed Graphs. In M.R. Sleep, M.J. Plasmeijer, and M.C. van Eekelen, editors, *Term Graph Rewriting: Theory and Practice*, chapter 14, pages 185–199. John Wiley, 1993.
11. Kurt Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching.* EATCS Monographs on Theoretical Computer Science. Springer, 1984.
12. Benjamin Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. *Journal of Mathematical Structures in Computer Science*, 6(5):409–454, 1996.
13. Detlef Plump. Hypergraph rewriting: Critical pairs and undecidability of confluence. In M.R. Sleep, M.J. Plasmeijer, and M.C. van Eekelen, editors, *Term Graph Rewriting: Theory and Practice*, chapter 15, pages 201–214. John Wiley, 1993.
14. Arend Rensink. Model checking graph grammars. In *Proc. of AVOCS '03 (Workshop on Automated Verification of Critical Systems)*, 2003.
15. Arend Rensink. Canonical graph shapes. In *Proc. of ESOP '04*, volume 2986 of *LNCS*, pages 401–415. Springer, 2004.
16. Arend Rensink. State space abstraction using shape graphs. In *Proc. of AVIS '04*, ENTCS, 2004. to appear.
17. Leila Ribeiro. *Parallel Composition and Unfolding Semantics of Graph Grammars.* PhD thesis, Technische Universität Berlin, 1996.
18. Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol.1: Foundations*, volume 1. World Scientific, 1997.
19. Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *TOPLAS (ACM Transactions on Programming Languages and Systems)*, 24(3):217–298, 2002.
20. Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proc. of POPL '96*. ACM, 1996.
21. Dejvuth Suwimonteerabuth, Stefan Schwoon, and Javier Esparza. jMoped: A Java Bytecode Checker Based on Moped. In *Proc. of TACAS 2005*, volume 3440 of *LNCS*, pages 541–545. Springer, 2005.
22. Dániel Varró. Towards symbolic analysis of visual modeling languages. In *Proc. of GT-VMT'02*, volume 72 of *ENTCS*. Elsevier, 2002.
23. Mandana Vaziri and Daniel Jackson. Checking properties of heap-manipulating procedures with a constraint solver. In *Proc. of TACAS 2003*, volume 2619 of *LNCS*, pages 505–520. Springer, 2003.
24. Robert Paul Wilson. *Efficient, Context-Sensitive Pointer Analysis for C Programs.* PhD thesis, Stanford University, 1997.

Local Reasoning for Termination

Andreas Podelski and Ina Schaefer

Max-Planck-Institut für Informatik, Saarbrücken, Germany

Abstract. In this paper, we bridge the gap between separation logic and transition invariants in order to obtain a uniform framework for proving total correctness of pointer programs. We introduce the concept of separated transition constraints to describe the local effect of pointer programs. Separated transition constraints provide a new view on locality by their non-tight interpretation. Furthermore, separated transition constraints constitute the basis for separated transition logic which enables local reasoning about relations over states of heap-manipulating programs. We put the framework of transition invariants to work with separated transition logic and obtain as a consequence a conservative extension of separation logic that is able to prove termination.

1 Introduction

So far, verification of pointer programs [12, 14] has mostly been restricted to safety properties. In this paper, we will present an approach for verifying pointer programs with respect to safety as well as liveness properties. While in this paper we will concentrate on termination the reduction to general liveness properties would follow the lines of [15, 6, 7].

We introduce separated transition constraints to describe the local effect of program statements in heap-manipulating programs. Separated transition constraints are transition constraints in primed and unprimed variables conjoined by $\wedge$ and $*$-conjunction. The spatial $*$-conjunction is borrowed from separation logic [12] and denotes that two parts of the heap must be disjoint.

Separated transition constraints offer a new viewpoint on locality in reasoning about program statements operating with pointers. Their footprint corresponds to the the minimal number of heap cells present in the heap for a valid interpretation. But this heap may be extended by an arbitrary number of unchanged heap cells. Separation logic requires an explicit frame rule which allows to extend a tight specifications over a heap to a larger heap area. For separated transition constraints, this is directly handled by the semantics. The frame axiom can simply be seen as a logical consequence thereof.

Separated transition constraints build the basis for separated transition logic which enables local reasoning about relations over states of heap-manipulating programs. This bridges the gap between the ideas of separation logic [12] where assertions only refer to single program states and transition invariants [7] where the proof rule is based on relations over program states.

Using separated transition logic we can put the framework of transition invariants [7] to work for heap manipulating programs and obtain a uniform proof rule for their total correctness. We will show that this proof rule is complete with respect to separation logic [12]. When it comes to termination arguments the proof rule is strictly more expressive. As a consequence, our results are a conservative extension of separation logic that is able to deal with termination.

Incorporating suitable abstraction techniques into the construction of separated transition constraints offers a high potential of the automatization of this proof rule. The results in transition predicate abstraction [8] offer a promising starting point in this direction.

This paper is structured as follows: In Section 2, we will introduce the concept of separated transition logic based on separated transition constraints. In Section 3, we will show how to use separated transition invariants for proving total correctness of pointer programs and illustrate this in Section 4 at the destructive reversal of a singly-linked list. In Section 5, we will briefly review some related work before concluding the paper in Section 6.

2 Separated Transition Logic

Separated transition logic facilitates local reasoning about relations of program states of a heap-manipulating program. It is based on the concept of separated transition constraints which describe the local effect of pointer programs.

Separated transition constraints are transition constraints in primed and unprimed variables conjoined by $\wedge$ and $*$-conjunction. For instance, the separated transition constraint

$$(x.n = x' \wedge x = y' \wedge x.n' = y)$$

expresses the transition corresponding to the body of the while loop of a program reversing a singly-linked list. In the heap consisting of at least one cell x is updated to its successor under its next field, y gets updated to the old value of x and the next field of the new value of x is updated to the old value of y.

Separated transition constraints are interpreted non-tightly over the heap. The footprint of a separated transition constraint denotes the heap area the constraint operates on. It corresponds to the minimal number of heap cells that must be present in the heap for a valid interpretation. But this heap may be extended by an arbitrary number of unchanged heap cells. Figure 1 depicts this situation for the above constraint. We use $n = n'$ as a shorthand notation that this heap part remains unchanged. We define atomic separated transition constraints as $v = w$, $v.n = w$ and $v.n' = w$, where v and w are primed and unprimed program variables, i.e. $v = x$ or $v = x'$ and $w = y$ or $w = y'$. Atomic separated transition constraints can be conjoined to separated transition constraints by $\wedge$- and $*$-conjunction. We use these constraints to relate the values $x, x.n$ and $x', x.n', x'.n'$ of the variable x and the next field n before resp. after a transition. The next field n is sometimes referred to as the cell content or the

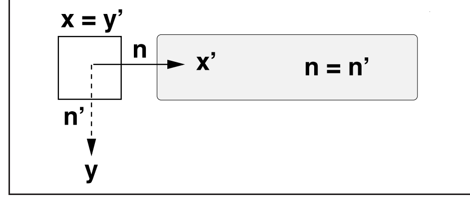

Fig. 1. Illustration of Separated Transition Constraint

value of the heap function. For notational convenience in the presence of primed variables, we use $x.n = y$ instead of $x \mapsto y$.

Definition 1 (Syntax of Separated Transition Constraints). *Let v be an element of the set of program variables. We define the syntax of tight separated transition constraints as*

$$\begin{aligned} w &::= v \mid v' \\ \varphi &::= emp \\ &\quad \mid w_1 = w_2 \\ &\quad \mid w_1.n = w_2 \\ &\quad \mid w_1.n' = w_2 \\ &\quad \mid \varphi_1 \ * \ \varphi_2 \\ &\quad \mid \varphi_1 \ \wedge \ \varphi_2 \end{aligned}$$

Separated transition constraints denote a relation over pairs of program states. A program state $\sigma \in \Sigma$ is a tuple consisting of a valuation $s \in S$ of the program variables and the heap function $h \in H$. The heap is modelled by a finite partial function mapping heap locations to values which themselves can be locations. In the presentation of this paper, we concentrate on just one successor function 'n' standing for the next-field of lists.

More formally, let $Loc \subseteq Val$ denote the set of heap locations where nil is a special heap location different from all other locations. Let Var denote a set of unprimed program variables. Let $s : Var \rightharpoonup_{fin} Val \in S$ be a finite partial function denoting an evaluation of the program variables and $h : Loc \rightharpoonup_{fin} Val \in H$ be a finite partial function denoting the heap function. Let $dom(h)$ denote its domain and $h_1 \uplus h_2$ that the two heap domains are disjoint.

The satisfaction relation $\models$ for separated transition constraints is defined inductively on the structure of the formula $c[Var, Var']$ in the standard way. For the spatial $*$-conjunction, we need a special definition which captures that the heap must be separable into two disjoint parts in each of one of the two conjuncts holds. Furthermore, the semantics definition allows the non-tight interpretation over a heap extended by an arbitrary number of unchanged cells.

Definition 2 (Semantics of Separated Transition Constraints).
Let σ and σ' be two program states. Then the satisfaction relation $\models$ for separated transition constraints is defined as follows:

$(\sigma,\sigma') \models \quad emp \quad$ *if* $dom(h) = \emptyset$

For $v_1, v_2 \in Var$:

$$
\begin{array}{lll}
(\sigma,\sigma') \models & (v_1 = v_2) & \text{if } s(v_1) = s(v_2) \\
 & \vdots & \vdots \\
(\sigma,\sigma') \models & (v_1' = v_2') & \text{if } s'(v_1) = s'(v_2) \\
(\sigma,\sigma') \models & (v_1.n = v_2) & \text{if } h(s(v_1)) = s(v_2) \text{ and } \{s(v_1)\} = dom(h) \\
 & \vdots & \vdots \\
(\sigma,\sigma') \models & (v_1'.n = v_2') & \text{if } h(s'(v_1)) = s'(v_2) \text{ and } \{s'(v_1)\} = dom(h) \\
(\sigma,\sigma') \models & (v_1.n' = v_2) & \text{if } h'(s(v_1) = s(v_2) \text{ and } \{s(v_1)\} = dom(h') \\
 & \vdots & \vdots \\
(\sigma,\sigma') \models & (v_1'.n' = v_2') & \text{if } h'(s'(v_1) = s'(v_2) \text{ and } \{s'(v_1)\} = dom(h')
\end{array}
$$

$$
\begin{array}{lll}
(\sigma,\sigma') \models & \varphi_1 * \varphi_2 & \text{if } h = h_1 \uplus h_2 \text{ and } h' = h_1' \uplus h_2' \text{ such that} \\
 & & \quad ((s,h_1),(s',h_1')) \models \varphi_1 \\
 & & \quad ((s,h_2),(s',h_2')) \models \varphi_2 \\
(\sigma,\sigma') \models & \varphi_1 \wedge \varphi_2 & \text{if } (\sigma,\sigma') \models \varphi_1 \text{ and } (\sigma,\sigma') \models \varphi_2
\end{array}
$$

Additionally, we define the non-tight interpretation

$$
\begin{array}{lll}
(\sigma,\sigma') \models & \varphi & \text{if } h = h_1 \uplus h_2 \text{ and } h' = h_1' \uplus h_2' \text{ and } h_2 = h_2' \text{ such that} \\
 & & \quad ((s,h_1),(s',h_1')) \models \varphi
\end{array}
$$

In the footprint of a separated transition constraint as defined by the syntax above, we collect all program variables that are required to be allocated by the constraint. The footprint is a syntactical construct because a separated transition constraint denotes a set of state pairs in which the domains of the heaps may differ by unchanged parts. So a separated transition constraint may be interpreted in heaps of different domains. However, if two constraints have the same footprint they share at least the same number of heap cells corresponding to the footprint and hence a subset relation must hold.

Definition 3 (Footprint). *We define the* footprint of a separated transition constraint φ *as*

$$footprint(\varphi) = \{x \in Var \mid \varphi \models \exists z.x.n = z \vee \exists z'.x.n' = z'\}$$

The frame axiom allows to extend a separated transition constraint by conjuncts that explicitly refer to unchanged areas of the heap. It corresponds to the frame rule used in separation logic and is a direct consequence of the semantics of separated transition constraints. It says that is possible to add a conjunct $x.n = x.n'$ to a constraint φ if x is not in the footprint of φ.

$$\varphi \equiv \varphi * (x.n = x.n') \text{ if } x \notin footprint(\varphi)$$

We define $x.n = x.n'$ as a shorthand notation for

$$\begin{aligned} &\exists z x.n = z \;\; \rightarrow (\exists z' x.n' = z' \wedge z = z') \\ \vee\; &\exists z' x.n' = z' \rightarrow \;\; (\exists z x.n = z \;\; \wedge z = z') \end{aligned}$$

This constraint requires that if a heap cell is allocated then it must remain unchanged. But it allows also the case in which x is not interpreted in the heap domain. This yields that for $\psi \stackrel{\text{def}}{=} \varphi * (x.n = x.n')$, the footprint of ψ does only comprise x if it is required to exist in the heap, e.g. by some other constraint.

For equivalence transformations, we use the following distributive law which states that the $\wedge$-conjunction distributes over the $*$-conjunction. It is a simple consequence of the definition of $*$-conjunction in separated transition constraints. It is used to conjoin two separated transition constraints working on two overlapping heap areas. It is applicable if φ_1 and ψ_1 have the same footprint.

$$(\varphi_1 \; * \; \varphi_2) \; \wedge \; (\psi_1 \; * \; \psi_2) \; \equiv \; (\varphi_1 \; \wedge \; \psi_1) \; * \; (\varphi_2 \; \wedge \; \psi_2)$$

The composition of two separated transition constraints $\varphi_1 \circ \varphi_2$ describes the relational composition of the denoted relations over states and heaps. This composition generalises strongest post and weakest preconditions. In the formula below, we rename the transition constraints (over the set of variables X and their primed versions; in our examples, $X = \{x, y, n\}$, $X' = \{x', y', n'\}$ and $X'' = \{x'', y'', n''\}$) and obtain a renamed version of the transition constraint for the composition. It is computed by

$$(\varphi_1 \circ \varphi_2)[X, X''] \equiv \exists X' \; (\varphi_1[X, X']) \wedge (\varphi_2[X', X''])$$

3 Proof Rule for Total Correctness

Having set up separated transition logic as framework for local reasoning over relations we are now ready to apply this to the proof rule proposed in [7] based on relations. Hence, we will now review the proof rule of [7] which applies to general programs, including pointer programs.

A program $\mathcal{P}$ is defined by a set of states $\Sigma \subseteq S \times H$, a set of designated starting states $I \subseteq \Sigma$ and a transition relation $R \subseteq \Sigma \times \Sigma$. We denote this transition relation by a set of transition constraints. A computation is a sequence of states $\sigma_1, \sigma_2, \ldots$ such that σ_1 is a starting state, i.e. $\sigma_1 \in I$ and for all $i \geq 1$ $(\sigma_i, \sigma_{i+1}) \in R$. A transition invariant is the superset of the transitive closure of the transition relation of the program restricted to the accessible states, i.e. formally

$$R^+ \cap (Acc \times Acc) \subseteq T$$

where $Acc = \{\sigma \in \Sigma \mid R^*(\sigma_1, \sigma), \sigma_1 \in I\}$. For every computation segment of the program, the start state and the end state are comprised in the transition invariant. In the proof rule, transition invariants are the only auxiliary assertion needed for proving termination and partial correctness of a program. An inductive relation T, i.e. $R \cup T \circ R \subset T$, is a transition invariant for $\mathcal{P}$.

If there are no infinite computations in a program the program terminates. This can be reduced to the well-foundedness of the transition invariant of the program. If T is a transition invariant for a program $\mathcal{P}$ that denotes a finite union of well founded relations then the program $\mathcal{P}$ terminates. The proof of this proposition can be found in [7] and is based on Ramsey's Theorem [11].

A program is partially correct with respect to a specification if it starts in an initial state satisfying the precondition and after completion satisfies the post-condition. If we restrict the transition invariant to the entry and the exit point of the program and add the precondition then the constraints in T must imply the post condition such that the program satisfies its specification. This proposition can be justified by remembering that the transition invariant contains the start and end state of every computation segment of the program and in particular state pairs for the entry and the exit point of the program.

We can now give the proof rule for total correctness of programs based on transition invariants.

Proof Rule

- $\mathcal{P} = (\Sigma, I, R)$ program
- $T \subseteq \Sigma \times \Sigma$ a relation over program states of $\mathcal{P}$
- pre and post pre and post condition for $\mathcal{P}$

1. $R \cup T \circ R \subseteq T$

2. $T \ \wedge \ \mathsf{pre} \ \wedge \ pc = entry \ \wedge \ pc' = exit \ \ \models \ \ \mathsf{post}$

3. T is a finite union of well founded relations.

$\mathcal{P}$ is totally correct

We now instantiate the setting above to separated transition constraints. This is straightforward since separated transition logic is designed to facilitate local reasoning over relations.

Definition 4 (Separated Transition Invariant). *A separated transition invariant is a set of separated transition constraints denoting a relation T over program states such that T is a transition invariant.*

We can show that separated transition logic, i.e. reasoning based on separated transition constraints, is complete relative to separation logic [12].

Theorem 1 (Relative Completeness with respect to Separation Logic). *Separated transition logic is complete relative to separation logic, i.e. if for a program $\mathcal{P}$ without allocation and deallocation of heap cells, separation logic proves the Hoare triple $\{\varphi_1\}\mathcal{P}\{\varphi_2\}$ correct then there exists a separated transition invariant denoting a relation T such that*

$$\varphi_1 \wedge T \models \varphi_2$$

Proof. The proof proceeds by structural induction over the set of axioms and inference rules of separation logic. Every axiom of separation logic can be translated to separated transition logic. Furthermore, we show that for each inference rule there exists a separated transition invariant that mimics the same reasoning step. We omit the detailed proof in the short version of this paper but give some examples:

– Mutation Axiom (Induction Basis) Separation logic proves

$$\{\exists z e.n = z\} e.n' = e_1 \{e.n' = e_1\}$$

Then $T \stackrel{\text{def}}{=} e.n' = e_1$ and hence $\varphi_1 \wedge T \models \varphi_2$.

– Frame Rule: (Induction Step) Separation logic proves

$$\frac{\{p\}c\{q\}}{\{p * r\}c\{q * r\}}$$

if r is not modified by c. Let $\psi = (x_1.n = x_1.n') * \ldots * (x_1.m = x_1.m')$ for all $x_i \in footprint(r)$. By induction hypothesis we know that there exists T such that $p \wedge T \models q$ and hence there exists T' with $T' \stackrel{\text{def}}{=} T * \psi$. Further, $T' \equiv T$ since $footprint(T) \cap footprint(r) = \emptyset$. Since $footprint(r) = footprint(\psi)$ the distributive law is applicable, thus

$$\begin{aligned} (p * r) \wedge T' &\models (p * r) \wedge (T * \psi) \\ &\models (p \wedge T) * (r \wedge \psi) \\ &\models (p \wedge T) * r \\ &\models (q * r) \end{aligned}$$

which yields the result. □

Furthermore, separated transition logic extends separation logic by termination arguments.

Theorem 2 (Relative Completeness with respect to Termination). *Separated transition logic is relatively complete for termination proofs, i.e. whenever a program $\mathcal{P}$ terminates there exists a separated transition invariant denoting a relation T such that T is a finite union of well-founded relations*

Proof. Thanks to our definition of separated transition constraints the proof in [7] can be directly extended. We omit the proof in the short version of the paper.

4 Case Example - List Reverse

In this section, we will illustrate how the first steps in our proof rule work for verifying total correctness of a destructive reversal on singly linked lists. We will establish that the relation denoted by a set of separated transition constraints indeed forms a 'valid' separated transition invariant. Figure 2 gives a pseudocode implementation of the considered program.

```
List reverse (List x)
{ List y,t;
  y := NULL;
l0 : while (x != NULL)
    {  t := y;
       y := x;
       x := x->n;
       y -> n := t;}
l1: return y;}
```

Fig. 2. Destructive List Reverse

For keeping track of program locations we will add a program counter to separated transition constraints. The program counter can be thought of as a special program variable ranging over the set of program locations only. Therefore, we add the conjuncts $pc = \ell_i$ and $pc' = \ell_j$ to a separated transition constraint in order to describe that this transition is only enabled if the program counter is ℓ_i before the transition and updated to ℓ_j afterwards.

The transition relation of the reverse program is the union $R \stackrel{\text{def}}{=} \{r_1, r_2\}$ of the two relations r_1 and r_2 over states. A state is a valuation of the variables x, y, n and pc (the program counter) where n is interpreted over the heap.

$$r_1 \stackrel{\text{def}}{\equiv} pc = l_0 \wedge c_1 \wedge pc' = l_0$$
$$r_2 \stackrel{\text{def}}{\equiv} pc = l_0 \wedge c_2 \wedge pc' = l_1$$

We can automatically translate the body of the while-loop to a separated transition constraint

$$d \equiv (x \neq nil \wedge x.n = x' \wedge x = y' \wedge y'.n' = y)$$

The precondition for the reverse program is that x points to an acyclic list. We define the predicate $list(x)$ as usual.

$$list(x) \equiv (emp \wedge x = nil) \ \vee \ \exists x_1 \ (x.n = x_1 * list(x_1))$$

Conjoining d with the precondition $list(x)$ yields that also after one execution of the while loop x points to an acyclic list, i.e. $d \wedge list(x) \models list(x')$. Hence it holds that the precondition distributes over the composition

$$(list(x) \wedge d) \circ d \equiv (list(x) \wedge d) \circ (list(x') \wedge d)$$

Therefore, the separated transition constraint c_1 corresponds to the translation of body of while-loop together with guard "x is an acyclic list", i.e. $c_1 \stackrel{\text{def}}{=} d \wedge list(x)$. Figure 3 illustrates the command c_1.

$$c_1 \stackrel{\text{def}}{\equiv} (x \neq nil \wedge x.n = x' \wedge x = y' \wedge y'.n' = y) * (list(x') \wedge n = n')$$

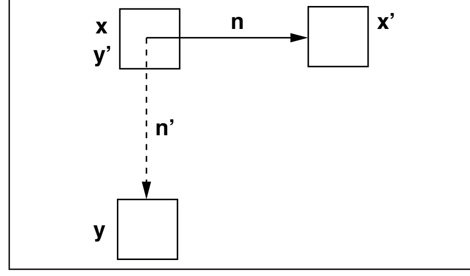

Fig. 3. Illustration of the body of the while loop

$list(x) \wedge (n = n')$ is a shorthand notation that the list remains unchanged. Formally,

$$list(x) \wedge (n = n') \stackrel{\text{def}}{\equiv} \begin{array}{l} (emp \wedge x = nil) \\ \vee\, (\exists x_1(x.n = x_1 \wedge x.n = x.n') * (list(x_1) \wedge (n = n')) \end{array}$$

The transition constraint c_2 corresponds to the termination of the while loop.

$$c_2 \stackrel{\text{def}}{\equiv} x = nil \wedge x' = x \wedge y' = y \wedge n = n'$$

The transitive closure T of the transition relation R can now be formulated as $T = r_1^+ \cup r_1^+ r_2 \cup r_2$ which also constitutes a separated transition invariant for the reverse program. By r_1^+, we denote the i times relational composition of the transition constraint r_1 with itself for $i \geq 1$ referring to i iterations of the while loop of the reverse program. By $r_1^+ r_2$, we denote the i times composition of r_1 with itself composed with the constraint r_2 corresponding to exiting the while-loop. We define two 'abstract' transition constraints for ${r_1}^+$ and $r_1^+ r_2$.

$$\begin{aligned} {r_1}^+ &\stackrel{\text{def}}{=} pc = l_0 \wedge \Phi \wedge pc' = l_0 \\ r_1^+ r_2 &\stackrel{\text{def}}{=} pc = l_0 \wedge \Psi \wedge pc' = l_1 \end{aligned}$$

In order to show that these abstract transition constraints indeed form a valid separated transition invariant, we will show that $\Phi \stackrel{\text{def}}{=} \bigvee_i \varphi_i \equiv c_1^+$ and that $\Psi \equiv c_1^+ c_2$.

The formula c_1^i denotes the i-times composition of the transition constraint c_1 with itself, for $i > 0$. We define φ_i as a shorthand for the conjunction below, we here omit the $\wedge$ and $*$ symbols.

$$\varphi_i \stackrel{\text{def}}{\equiv} \left(\begin{array}{rcl} x.n^{(i-1)} & \neq & nil \\ x.n' & = & y \\ x.n^{(i-1)} & = & y' \\ y'.n & = & x' \\ y'.n'^{(i-1)} & = & x \\ \bigwedge_{0 \leq j \leq i-1} y'.n'^{j} & = & x.n^{(i-1-j)} \end{array} \right) * \ (list(x') \wedge n = n')$$

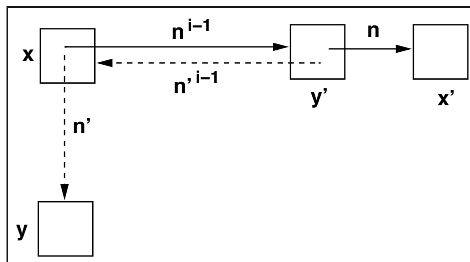

Fig. 4. Illustration of the first part of φ_i

This constraint models that the first i cells of the initial list x are changed, while the rest of the list still remains unchanged. The big conjunction expresses that the links between the first i list elements are already reversed. Figure 4 illustrates the first part of the formula φ_i and Figure 5 the big conjunction. The formula $x.n^i = z$ expresses that x and z are linked by an acyclic chain of i next fields. Formally,

$$x.n^0 = z \equiv x = z$$
$$x.n^{(i+1)} = z \equiv \exists y\ (x.n = y * y.n^i = z)$$

In the definition of φ_i, the conjunct $x.n^{(i-1)} = y'$ is the special case of the big conjunction for $j = 0$, and the conjunct $y'.n'^{(i-1)} = x$ the one for $j = i - 1$.

The crucial step in the total correctness proof is to show that the i-times composition of the transition constraint c_1 translating the guarded body of the while of reverse is equivalent to the formula φ_i. The equivalence is proven by induction over i. The base case for $i = 1$ follows by definition. Below we spell out the proof of the equivalence for the case $i = 2$. This is sufficient to give us the idea of the general induction step (which is analogous but more unpleasant to read). Figure 6 illustrates the case for $i = 2$, i.e. $c_1 \circ c_1[var, var', var'']$. The result can then be derived as depicted in Figure 7.

In the presentation of Figure 7 we omit several details. In particular, we do not include the (elimination of the) existential quantifiers of single-primed variables. At step (1) we use the frame axiom where $x \in \mathit{footprint}(c_1[X, X'])$

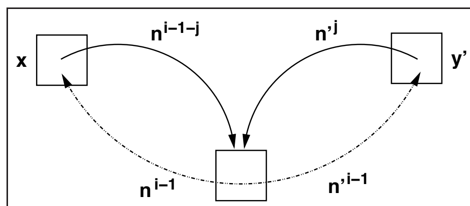

Fig. 5. Illustration of the big conjunction in φ_i

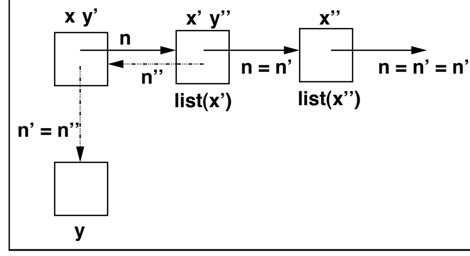

Fig. 6. Illustration of $c_1 \circ c_1$

but $x \notin \backslash footprint(c_1[X', X''])$. Then we are able to apply the distributive law in step (2) because the two first cells of the list have the same footprint x. Additionally, we use the equivalence

$$list(x') \wedge n = n' \quad \equiv \quad (x'.n = x'' \wedge x'.n = x'.n') * (list(x'') \wedge n = n').$$

Finally in step (3), we apply the distributive law again for the two second cells which have the same footprint x'. The big conjunction appearing in φ_i does not appear in our presentation of φ_2. The conjunct $x.n = y'$ is the special case of the big conjunction for $i = 2$ and $j = 0$, and the conjunct $y'.n' = x$ the one for $i = 2$ and $j = i - 1$ (and those two conjuncts form φ_2).

We define Ψ as a shorthand for the conjunction below, we again omit the $\wedge$ and $*$ symbols.

$$\Psi \stackrel{\text{def}}{\equiv} \begin{pmatrix} k & > & 0 \\ x.n^{(k-1)} & \neq & nil \\ x.n' & = & y \\ x.n^{(k-1)} & = & y' \\ y'.n & = & nil \\ y'.n'^{(k-1)} & = & x \\ \bigwedge_{0 \le j \le k-1} \quad x.n^j & = & y'.n'^{(k-1-j)}) \end{pmatrix}$$

We need to show that the composition $c_1^i \circ c_2$ of the i-times iteration of the body of the while loop with the exit transition c_2 is equivalent to the the formula Ψ, for $i = k$ and x different from the empty list. We omit the proof of this equivalence; it is analogous to the proof of $c_1^i \equiv \varphi_i$.

Now, we can reason about total correctness of the reverse program from its separated transition invariant. The program terminates because T denotes a finite union of well-founded relations. We show that r_1^+, $r_1^+ r_2$ and r_2 each are well founded relations, i.e. there are no infinite chains. This is trivially true for $r_1^+ r_2$ and r_2 since the values of the program counter before and after the transition are different, i.e. there are no chains of length greater than 2. The length of a chain of r_1^+-transitions that starts in a state σ is bounded by the length of the list x in the state σ, since r_1^+ entails $x.n^i = x'$, i.e. the length of the list x decreases

$$(c_1 \circ c_1)[X, X'']$$

$$\equiv \quad c_1[X, X'] \quad \wedge \quad c_1[X', X'']$$

$$\equiv \left(\begin{array}{c} \hline x \neq nil \\ x.n = x' \\ x = y' \\ x.n' = y \end{array} * \begin{array}{c} \hline list(x') \\ n = n' \end{array} \right) \wedge \left(\begin{array}{c} \hline x' \neq nil \\ x'.n' = x'' \\ x' = y'' \\ x'.n'' = y' \end{array} * \begin{array}{c} \hline list(x'') \\ n' = n'' \end{array} \right)$$

$$\stackrel{(1)}{\equiv} \left(\begin{array}{c} \hline x \neq nil \\ x.n = x' \\ x = y' \\ x.n' = y \end{array} * \begin{array}{c} \hline list(x') \\ n = n' \end{array} \right) \wedge \left(\begin{array}{c} \hline x.n' = x.n'' \end{array} * \begin{array}{c} \hline x' \neq nil \\ x'.n' = x'' \\ x' = y'' \\ x'.n'' = y' \end{array} * \begin{array}{c} \hline list(x'') \\ n' = n'' \end{array} \right)$$

$$\stackrel{(2)}{\equiv} \left(\begin{array}{c} \hline x \neq nil \\ x.n = x' \\ x = y' \\ x.n' = y \\ x.n' = x.n'' \end{array} * \left[\left(\begin{array}{c} \hline x'.n = x'' \\ x'.n = x'.n' \end{array} * \begin{array}{c} \hline list(x'') \\ n = n' \end{array} \right) \wedge \left(\begin{array}{c} \hline x' \neq nil \\ x'.n' = x'' \\ x' = y'' \\ x'.n'' = y' \end{array} * \begin{array}{c} \hline list(x'') \\ n' = n'' \end{array} \right) \right] \right)$$

$$\stackrel{(3)}{\equiv} \left(\begin{array}{c} \hline x \neq nil \\ x.n = x' \\ x = y' \\ x.n' = y \\ x.n' = x.n'' \end{array} * \begin{array}{c} \hline x' \neq nil \\ x'.n' = x'' \\ x' = y'' \\ x'.n'' = y' \\ x'.n = x'.n' \end{array} * \begin{array}{c} \hline list(x'') \\ n = n' = n'' \end{array} \right)$$

$$\equiv \quad \varphi_2[X, X'']$$

Fig. 7. Derivation of $c_1^2 \equiv \varphi_2$

by $i > 0$ with each r_1-transition in the chain. So for $i \geq 1$, r_1^+ is contained in a well founded relation,

$$r_1^+ \models (x.n^{(i-1)} \neq nil \wedge x.n^i = x') * (list(x') \wedge n = n')$$

Partial correctness follows from the fact that the separated transition invariant T restricted to the pair of initial resp. final program locations entails the pre and post condition pair.

$$T \wedge (pc = l_0) \wedge (pc' = l_1) \wedge \mathsf{pre} \models \mathsf{post}$$

The precondition is that x is an acyclic list. The postcondition is that the program implements an in-place reversal of the list x. The new value of y points successively to the elements of the list x, in reverse order. The correctness of the reverse program does not depend on having the initial value of y being nil or being a list in a disjoint part of the heap. Our correctness criterion is strictly stronger than the reversal of the word represented by the list.

$$\mathsf{pre} \stackrel{\text{def}}{\equiv} list(x)$$

$$\begin{aligned}\mathsf{post} \stackrel{\text{def}}{\equiv} & \quad length(x) = 0 \wedge x' = x \wedge y' = y \wedge n' = n \\ & \vee \textstyle\bigwedge_{0 \leq j < length(x)} \quad x.n^j = y'.n'^{\,length(x)-1-j}\end{aligned}$$

We observe that $T \wedge (pc = l_0) \wedge (pc' = l_1)$ is equivalent to $\{r_1^+ r_2, r_2\}$. The transition constraint r_2 entails the first disjunct of the post condition, and $r_1^+ r_2$ the second.

5 Related Work

For comparison with existing methods to prove partial correctness of pointer programs, separation logic [12] is a Hoare style logic where separation logic assertions are used for specifying partial correctness. The main feature of separation logic is the spatial conjunction operator $*$ which explicitly expresses separation of two parts of the heap. Separated transition constraints borrow this spatial conjunction operator. In contrast to the pre and post condition pairs of separation logic, a single separated transition constraint can be used to describe the effect of a program statement. The Hoare triples in separation logic are generally tight and speak only about the memory area and variables currently used. The frame rule allows to extend a local specification by adding arbitrary variables and predicates not modified by commands in the Hoare triple. In our approach, this explicit frame rule is directly encoded into the interpretation of separated transition constraints and into the frame axiom. Separation logic so far is unable to deal with termination or general liveness properties while in this paper we have shown that separated transition logic provides a conservative extension of separation logic that is able to deal with termination arguments.

The pointer assertion logic engine (PALE) [5] is another framework for verifying partial correctness of programs dealing with pointer structures that can be expressed as graph types in second order monadic logic. However, this framework is also not able to deal with reasoning about termination and other liveness properties.

Another well-known approach to automatically prove safety properties of pointer programs is shape analysis [14] based on abstract interpretation. The properties shown by this framework are safety properties in the flavour of no nil-pointer dereference, no memory leaks or structural invariance properties. Shape analysis abstracts the structures in the heap according to a chosen set of abstraction predicates. This abstraction, however, prevents to reason about liveness properties. There are several approaches to extend classical shape analysis by means to reason about recursive pointer programs [13, 4] or to transfer predicate abstraction techniques [3] to shape analysis [2, 10]. However, none of these is able to deal with termination reasoning.

The only approach known to us dealing with liveness properties of pointer programs is [1]. The authors propose to abstract a heap transition system and the property to be verified by an abstraction preserving the desired properties of the concrete system. Liveness properties are handled by adding a well-founded ranking function to the system. This function is transformed into a progress monitor which is composed with the system under consideration and subsequently abstracted.

6 Discussion

In this paper, we have presented a novel approach to the verification of total correctness of pointer programs which bridges the gap between ideas from separation logic and transition invariants.

However, some problems have not been solved completely yet. Separated transition logic is at the moment unable to deal with allocation and deallocation of memory cells. These operations still have to be integrated into our framework. Furthermore, the concept of footprint is problematic in the presence of disjunctions of separated transition constraints. Consider the constraint

$$\varphi \stackrel{\text{def}}{=} x.n = y \wedge ((y = nil) \vee (\exists z\ y.n = z))$$

which occurs for example using $list(x)$. Then $y \notin footprint(\varphi)$ because by the first disjunct $y = nil$, y is not allocated in the heap. But frame axiom is not applicable because the second disjunct $\exists z\ y.n = z$ requires that y is already allocated and hence $\varphi * (y.n = y.n')$ is inconsistent. Additionally, the definition of footprints for predicates like the list predicate which require a previously unknown number of heap cells is an open question. While this is not a problem in the presentation of this paper the concept of footprint needs more subtle investigations.

For the future, we aim at adding suitable abstraction techniques, as in [2, 13, 14, 10], to the concept of separated transition constraints in order to automatize

the proof rule. The results in transition predicate abstraction [8] can be transferred to this setting and offer a promising starting point for the development of an automatic method for the verification of pointer programs. Furthermore, we want to extend the approach presented here to handle recursive pointer programs by using the generalised notion of procedure summaries, as pointed out in [9].

Acknowledgements. The authors would like to thank Peter O'Hearn, Christiano Calcagno and Hongseok Yang for comments and suggestions.

References

1. I. Balaban, L. Zuck, and A. Pnueli. Shape analysis by predicate abstraction. In *Proceedings to VMCAI05*, 2005.
2. Dennis Dams and Kedar S. Namjoshi. Shape Analysis through Predicate Abstraction and Model Checking. In *Proceedings of the 4th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'03)*, volume 2575 of *Lecture Notes in Computer Science*, pages 310–323. Springer-Verlag, 2003.
3. Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In *Proceedings of CAV'1997: Computer Aided Verification*.
4. B. Jeannet, A. Loginov, T. Reps, and M. Sagiv. A relational approach to interprocedural shape analysis. In *Proceedings of SAS'04*, 2004.
5. A. Moeller and M. Schwartzbach. The pointer assertion logic engine. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation, PLDI'01*, 2001.
6. A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In *Proceedings of VMCAI'04*, 2004.
7. A. Podelski and A. Rybalchenko. Transition invariants. In *Proc. of LICS'2004: Logic in Computer Science*, pages 32–41. IEEE, 2004.
8. A. Podelski and A. Rybalchenko. Transition predicate abstraction and fair termination. In *Proc. of POPL'2005: Principles of Programming Languages*. ACM Press, 2005.
9. A. Podelski, I. Schaefer, and S. Wagner. Summaries for while programs with recursion. *Proceedings of European Symposium on Programing (ESOP'05)*, 2005.
10. A. Podelski and Th. Wies. Boolean heaps. to appear in SAS'05, 2005.
11. F. P. Ramsey. On a problem of formal logic. In *Proceedings London Math. Soc.*, 1930.
12. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. *Proceedings of LICS'02*, 2002.
13. Noam Rinetzky and Mooly Sagiv. Interprocedural shape analysis for recursive programs. *Lecture Notes in Computer Science*, 2027, 2001.
14. Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(3):217–298, 2002.
15. M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of LICS'86*, 1986.

Symbolic Execution with Separation Logic

Josh Berdine[1], Cristiano Calcagno[2], and Peter W. O'Hearn[1]

[1] Queen Mary, University of London
[2] Imperial College, London

Abstract. We describe a sound method for automatically proving Hoare triples for loop-free code in Separation Logic, for certain preconditions and postconditions (symbolic heaps). The method uses a form of symbolic execution, a decidable proof theory for symbolic heaps, and extraction of frame axioms from incomplete proofs. This is a precursor to the use of the logic in automatic specification checking, program analysis, and model checking.

1 Introduction

Separation Logic has provided an approach to reasoning about programs with pointers that often leads to simpler specifications and program proofs than previous formalisms [9]. This paper is part of a project attempting to transfer the simplicity of the by-hand proofs to a fully automatic setting.

We describe a method for proving Hoare triples for loop-free code, by a form of symbolic execution, for certain (restricted) preconditions and postconditions. It not our intention here to try to show that the method is useful, just to say what it is, and establish its soundness. This is a necessary precursor to further possible developments on using Separation Logic in:

- *Automatic Specification Checking*, where one takes an annotated program (with preconditions, postconditions and loop invariants) and chops it into triples for loop-free code in the usual way;
- *Program Analysis*, where one uses fixed-point calculations to remove or reduce the need for annotations; and
- *Software Model Checking*.

The algorithms described here are in fact part an experimental tool of the first variety, Smallfoot. Smallfoot itself is described separately in a companion paper [2]; here we confine ourselves to the technical problems lying at its core. Of course, program analysis and model checking raise further problems – especially, the structure of our "abstract" domain and the right choice of widening operators [3] – and further work is under way on these.

There are three main issues that we consider.

1. How to construe application of Separation Logic proof rules as symbolic execution. The basic idea can be seen in the axiom

$$\{A * x{\mapsto}[f\colon y]\}\ x{\to}f{:=}\,z\ \{A * x{\mapsto}[f\colon z]\}$$

where the precondition is updated in-place, in a way that mirrors the imperative update of the actual heap that occurs during program execution. The separating conjunction, $*$, short-circuits the need for a global alias check in this axiom. $A * x{\mapsto}[f\colon y]$ says that the heap can be partitioned into a single cell x, that points to (has contents) a record with y in its f field, and the rest of the heap, where A holds. We know that A will continue to hold in the rest of the heap if we update x, because x is not in A's part of the heap.

There are two restrictions on assertions which make the symbolic execution view tenable. First, we restrict to a format of the form $B \wedge S$ where B is a pure boolean formula and S is a $*$-combination of heap predicates. We think of these assertions as "symbolic heaps"; the format makes the analogy with the in-place aspect of concrete heaps apparent. Second, the preconditions and postconditions do not describe the detailed contents of data structures, but rather describe shapes (in roughly the sense of the term used in shape analysis). Beyond the basic primitives of Separation Logic Smallfoot at this point includes several hardwired shape predicates: for singly- and doubly-linked lists, for xor-linked lists, and for trees. Here we describe our results for singly-linked lists and trees only.

2. How to discharge entailments $A \vdash B$ between symbolic heaps. We give a decidable proof theory for the assertions in our language.

One key issue is how to account for entailments that would normally require induction. To see the issue, consider a program for appending two lists. When you get to the end of the first list you link it up to the second. At this point to prove the program requires showing an entailment

$$\mathsf{ls}(x,t) * t{\mapsto}[n\colon y] * \mathsf{ls}(y,\mathsf{nil}) \vdash \mathsf{ls}(x,\mathsf{nil})$$

where we have a list segment from x to t, a single node t, and a further segment (the second list) from y up to nil. The entailment itself does not follow at once from simple unwinding of an inductive definition of list segments. In the metatheory it is proven by induction, and in our proof theory it will be handled using rules that are consequences of induction but that are themselves non-inductive in character.

In [1] we showed decidability of a fragment of the assertion language of this paper, concentrating on list segments. Here we give a new proof procedure, which appears to be less fragile in the face of extension than the model-theoretic procedure of [1]. Additionally, and crucially, it supports inference of frame axioms.

3. Inference of Frame Axioms. Separation Logic allows specifications to be kept small because of it avoids the need to state frame axioms, which describe the portions of heap not altered by a command [8]. To see the issue, suppose you have a specification

$$\{\mathsf{tree}(p)\}\ \mathtt{DispTree}(p)\ \{\mathsf{emp}\}$$

for disposing a tree, which just says that if you have a tree (and nothing else) and you dispose it, then there is nothing left. When verifying a recursive procedure for disposing a tree there will be recursive calls for disposing subtrees. The

problem is that, generally, a precondition at a call site will not match that for the procedure because there will be extra heap around. For example, at the site of a call $\texttt{DispTree}(i)$ to dispose the left subtree we might have a root pointer p and the right subtree j as well as the left subtree – $p{\mapsto}[l\colon i, r\colon j] * \mathsf{tree}(i) * \mathsf{tree}(j)\}$ – while the precondition for the overall procedure specification expects only a single tree.

Separation Logic has a proof rule, the Frame Rule, which allows us to resolve this mismatch. It allows us the first step in the inference:

$$\dfrac{\dfrac{\{\mathsf{tree}(i)\}\texttt{DispTree}(i)\{\mathsf{emp})\}}{\{p{\mapsto}[l\colon i, r\colon j] * \mathsf{tree}(i) * \mathsf{tree}(j)\}\texttt{DispTree}(j)\{p{\mapsto}[l\colon i, r\colon j] * \mathsf{emp} * \mathsf{tree}(j)\}}}{\{p{\mapsto}[l\colon i, r\colon j] * \mathsf{tree}(i) * \mathsf{tree}(j)\}\texttt{DispTree}(j)\{p{\mapsto}[l\colon i, r\colon j] * \mathsf{tree}(j)\}}$$

To automatically generate proof steps like this we need some way to infer frame axioms, the leftover parts (in this case $p{\mapsto}[l\colon i, r\colon j] * \mathsf{tree}(j)$). Sometimes, this leftover part can be found by simple pattern matching, but often not. In this paper we describe a novel method of extracting frame axioms from incomplete proofs in our proof theory for entailments. A failed proof can identify the "left-over" part which, were you to add it in, would complete the proof, and we show how this can furnish us with a sound choice of frame axiom.

The notion of symbolic execution presented in this paper is, in a general sense, similar in spirit to what one obtains in Shape Analysis or PALE [11, 7]. However, there are nontrivial differences in the specifics. In particular, we have been unsuccessful in attempts to compositionally translate Separation Logic into either PALE's assertion language or into a shape analysis; the difficulty lies in treating the separating conjunction connective. And this is the key to employing the frame rule, which is responsible for Separation Logic's small specifications of procedures. So it seems sensible to attempt to describe symbolic execution for Separation Logic directly, in its own terms. (Of course, once this is done we hope that detailed comparisons and even Nelson-Oppen style coroutining of proof methods will then be possible.)

2 Symbolic Heaps

The concrete heap models assume:

- A fixed finite collection Fields;
- Disjoint sets Loc of locations, Val of non-addressable values, with $\mathsf{nil} \in \mathrm{Val}$.

We then set:

$$\mathrm{Heaps} \stackrel{\mathrm{def}}{=} \mathrm{Loc} \stackrel{\mathrm{fin}}{\rightharpoonup} (\mathrm{Fields} \to \mathrm{Val} \cup \mathrm{Loc})$$

$$\mathrm{Stacks} \stackrel{\mathrm{def}}{=} \mathrm{Var} \to \mathrm{Val} \cup \mathrm{Loc}$$

As a language for talking about these models we consider certain pure (heap independent) and spatial (heap dependent) assertions.

$x, y, \ldots \in \mathrm{Var}$ variables

$$\begin{array}{rll}
E & ::= \mathsf{nil} \mid x & \text{expressions} \\
P & ::= E{=}E \mid \neg P & \text{simple pure formulæ} \\
\Pi & ::= \mathsf{true} \mid P \mid \Pi \wedge \Pi & \text{pure formulæ} \\
f, f_i, \ldots & \in \text{Fields} & \text{fields} \\
\rho & ::= f_1{:}\, E_1, \ldots, f_k{:}\, E_k & \text{record expressions} \\
S & ::= E{\mapsto}[\rho] & \text{simple spatial formulæ} \\
\Sigma & ::= \mathsf{emp} \mid S \mid \Sigma * \Sigma & \text{spatial formulæ} \\
& \Pi \mathrel{\vert} \Sigma & \text{symbolic heaps}
\end{array}$$

The pure part here is oriented to stating facts about pointer programs, where we will use equality with nil to indicate a situation where we do not have a pointer. Other subsets of boolean logic could be considered in other situations.

In this heap model a location maps to a record of values. The formula $E{\mapsto}[\rho]$ can mention any number of fields in ρ, and the values of the remaining fields are implicitly existentially quantified. This allows us to write specifications which do not mention fields whose values we do not care about.

The semantics is given by a forcing relation $s, h \vDash A$ where $s \in$ Stacks, $h \in$ Heaps, and A is a pure assertion, spatial assertion, or symbolic heap.

$$[\![x]\!]s \stackrel{\text{def}}{=} s(x) \qquad\qquad [\![\mathsf{nil}]\!]s \stackrel{\text{def}}{=} \text{nil}$$

$$\begin{array}{lll}
s, h \vDash E_1{=}E_2 & \stackrel{\text{def}}{\text{iff}} & [\![E_1]\!]s = [\![E_2]\!]s \\
s, h \vDash \neg P & \stackrel{\text{def}}{\text{iff}} & s, h \nvDash P \\
s, h \vDash \mathsf{true} & & \text{always} \\
s, h \vDash \Pi_0 \wedge \Pi_1 & \stackrel{\text{def}}{\text{iff}} & s, h \vDash \Pi_0 \text{ and } s, h \vDash \Pi_1 \\
s, h \vDash E_0{\mapsto}[f_1{:}\, E_1, \ldots, f_k{:}\, E_k] & \stackrel{\text{def}}{\text{iff}} & h = [[\![E_0]\!]s{\to}r] \text{ where } r(f_i) = [\![E_i]\!]s \text{ for } i \in 1..k \\
s, h \vDash \mathsf{emp} & \stackrel{\text{def}}{\text{iff}} & h = \varnothing \\
s, h \vDash \Sigma_0 * \Sigma_1 & \stackrel{\text{def}}{\text{iff}} & \exists h_0 h_1.\, h = h_0 {*} h_1 \text{ and } s, h_0 \vDash \Sigma_0 \text{ and } s, h_1 \vDash \Sigma_1 \\
s, h \vDash \Pi \mathrel{\vert} \Sigma & \stackrel{\text{def}}{\text{iff}} & s, h \vDash \Pi \text{ and } s, h \vDash \Sigma
\end{array}$$

Note that we abbreviate $\neg(E_1{=}E_2)$ as $E_1{\neq}E_2$ and $\mathsf{true} \mathrel{\vert} \Sigma$ as Σ, and use $\equiv$ to denote "syntactic" equality of formulæ, which are considered up to symmetry of $=$, permutations across $\wedge$ and $*$, *e.g.*, $\Pi \wedge P \wedge P' \equiv \Pi \wedge P' \wedge P$, involutivity of negation, and unit laws for true and emp. We use notation treating formulæ as sets of simple formulæ, *e.g.*, writing $P \in \Pi$ for $\Pi \equiv P \wedge \Pi'$ for some Π'.

To reason about pointer programs one typically needs predicates that describe inductive properties of the heap. We describe two of the predicates (adding to the simple spatial formulæ) that we have experimented with in Smallfoot.

2.1 Trees

We describe a model of binary trees where each internal node has fields l, r for the left and right subtrees. The empty tree is given by nil. What we require is

that $\mathsf{tree}(E)$ is the least (logically strongest) predicate satisfying:

$$\begin{aligned}\mathsf{tree}(E) \iff & (E = \mathsf{nil} \wedge \mathsf{emp}) \\ & \vee (\exists x, y.\, E{\mapsto}[l\colon x, r\colon y] \,*\, \mathsf{tree}(x) \,*\, \mathsf{tree}(y))\end{aligned}$$

where x and y are fresh. The use of the $*$ between $E{\mapsto}[l\colon x, r\colon y]$ and the two subtrees ensures that there are no cycles, and the $*$ between the subtrees ensures that there is no sharing; it is not a DAG.

The way that the record notation works allows this definition to apply to any heap model that contains at least l and r fields. In case there are further fields, say a field d for the data component of a node, the definition is independent of what the specific values are in those fields.

Our description of this predicate is not entirely formal, because we do not have existential quantification, disjunction, or recursive definitions in our fragment. However, what we are doing is defining a new simple spatial formula (extending syntactic category S above), and we are free to do that in the metatheory. A longer-winded way to view this, as a semantic definition, is to say that it is the least predicate such that

$s, h \vDash \mathsf{tree}(E)$ holds if and only if
1. $s, h \vDash E = \mathsf{nil} \wedge \mathsf{emp}$, or
2. there are ℓ_x, ℓ_y where

$$(s \mid x \mapsto \ell_x, y \mapsto \ell_y), h \vDash E{\mapsto}[l\colon x, r\colon y] \,*\, \mathsf{tree}(x) \,*\, \mathsf{tree}(y)$$

Of course, we would have to prove (in the metatheory) that the least definition exists, but that is not difficult.

2.2 List Segments

We will work with linked lists that use field n for the next element. The predicate for linked list segments is the least satisfying the following specification:

$$\begin{aligned}\mathsf{ls}(E, F) \iff & (E{=}F \wedge \mathsf{emp}) \\ & \vee (E{\neq}F \wedge \exists y.E{\mapsto}[n\colon y] \,*\, \mathsf{ls}(y, F))\end{aligned}$$

Once again, this definition allows for additional fields, such as a head field, but the ls predicate is insensitive to the values of these other fields.

With this definition a complete linked list is one that satisfies $\mathsf{ls}(E, \mathsf{nil})$. Complete linked lists, or trees for that matter, are much simpler than segments. But the segments are sometimes needed when reasoning in the middle of a list, particularly for iterative programs. (Similar remarks would apply to tree segments.)

3 Symbolic Execution

In this section we give rules for triples of the form

$$\{\Pi \mid \Sigma\}\; C\; \{\Pi' \mid \Sigma'\}$$

where C is a loop-free program. The commands C are given by the grammar:

$$
\begin{array}{lll}
C ::= & \mathtt{empty} & \text{empty command} \\
\mid & x := E \; ; \; C & \text{variable assignment} \\
\mid & x := E{\rightarrow}f \; ; \; C & \text{heap lookup} \\
\mid & E{\rightarrow}f := F \; ; \; C & \text{heap mutation} \\
\mid & \mathtt{new}(x) \; ; \; C & \text{allocation} \\
\mid & \mathtt{dispose}(E) \; ; \; C & \text{disposal} \\
\mid & \mathtt{if}\, P \,\mathtt{then}\, C \,\mathtt{else}\, C \,\mathtt{fi} \; ; \; C & \text{conditional}
\end{array}
$$

The rules in this section appeal to entailments $\Pi \mid \Sigma \vdash \Pi' \mid \Sigma'$ between symbolic heaps. Semantically, entailment is defined by:

$$\Pi \mid \Sigma \vdash \Pi' \mid \Sigma' \text{ is true} \quad \text{iff} \quad \forall s, h.\, s, h \vDash \Pi \mid \Sigma \text{ implies } s, h \vDash \Pi' \mid \Sigma'$$

For the presentation of rules in this section we will regard semantic entailment as an oracle. Soundness of symbolic execution just requires an approximation.

3.1 Operational Rules

The operational rules use the following notation for record expressions:

$$mutate(\rho, f, F) = \begin{cases} f{:}\, F, \rho' & \text{if } \rho = f{:}\, E, \rho' \\ f{:}\, F, \rho & \text{if } f \notin \rho \end{cases}$$

$$lookup(\rho, f) = \begin{cases} E & \text{if } \rho = f{:}\, E, \rho' \\ x \text{ fresh} & \text{if } f \notin \rho \end{cases}$$

The fresh variable returned in the lookup case corresponds to the idea that if a record expression does not give a value for a particular field then we do not care what it is. These definitions do not result in conditionals being inserted into record expressions; they do not depend on the values of variables or the heap.

The operational rules are shown in Table 1. One way to understand these rules is by appeal to operational intuition. For instance, reading bottom-up, from conclusion to premise, the Mutate rule says:

> To determine if $\{\Pi \mid \Sigma * E{\mapsto}[\rho]\}\ E{\rightarrow}f := F \; ; \; C\ \{\Pi' \mid \Sigma'\}$ holds, execute $E{\rightarrow}f := F$ on the symbolic pre-state $\Pi \mid \Sigma * E{\mapsto}[\rho]$, updating E in place, and then continue with C.

Likewise, the Dispose rule says to dispose a symbolic cell (a $\mapsto$ fact), the New rule says to allocate, and the Lookup rule to read. The substitutions of fresh variables are used to keep track of (facts about) previous values of variables.

The role of fresh variables can be understood in terms of standard considerations on Floyd-Hoare logic. Recall that in Floyd's assignment axiom

$$\{A\}\ x := E\ \{\exists x'.\, x{=}E[x'/x] \wedge A[x'/x]\}$$

Table 1 Operational Symbolic Execution Rules

EMPTY

$$\frac{\Pi \mid \Sigma \vdash \Pi' \mid \Sigma'}{\{\Pi \mid \Sigma\}\ \texttt{empty}\ \{\Pi' \mid \Sigma'\}}$$

ASSIGN

$$\frac{\{x{=}E[x'/x] \wedge (\Pi \mid \Sigma)[x'/x]\}\ C\ \{\Pi' \mid \Sigma'\}}{\{\Pi \mid \Sigma\}\ x{:=}E\ ;\ C\ \{\Pi' \mid \Sigma'\}}\ x' \text{ fresh}$$

LOOKUP

$$\frac{\{x{=}F[x'/x] \wedge (\Pi \mid \Sigma * E{\mapsto}[\rho])[x'/x]\}\ C\ \{\Pi' \mid \Sigma'\}}{\{\Pi \mid \Sigma * E{\mapsto}[\rho]\}\ x{:=}E{\to}f\ ;\ C\ \{\Pi' \mid \Sigma'\}}\ x' \text{ fresh}, \mathit{lookup}(\rho, f) = F$$

MUTATE

$$\frac{\{\Pi \mid \Sigma * E{\mapsto}[\rho']\}\ C\ \{\Pi' \mid \Sigma'\}}{\{\Pi \mid \Sigma * E{\mapsto}[\rho]\}\ E{\to}f{:=}F\ ;\ C\ \{\Pi' \mid \Sigma'\}}\ \mathit{mutate}(\rho, f, F) = \rho'$$

NEW

$$\frac{\{(\Pi \mid \Sigma)[x'/x] * x{\mapsto}[]\}\ C\ \{\Pi' \mid \Sigma'\}}{\{\Pi \mid \Sigma\}\ \texttt{new}(x)\ ;\ C\ \{\Pi' \mid \Sigma'\}}\ x' \text{ fresh}$$

DISPOSE

$$\frac{\{\Pi \mid \Sigma\}\ C\ \{\Pi' \mid \Sigma'\}}{\{\Pi \mid \Sigma * E{\mapsto}[\rho]\}\ \texttt{dispose}(E)\ ;\ C\ \{\Pi' \mid \Sigma'\}}$$

CONDITIONAL

$$\frac{\{\Pi \wedge P \mid \Sigma\}\ C_1\ ; C\ \{\Pi' \mid \Sigma'\} \qquad \{\Pi \wedge \neg P \mid \Sigma\}\ C_2\ ; C\ \{\Pi' \mid \Sigma'\}}{\{\Pi \mid \Sigma\}\ \texttt{if}\ P\ \texttt{then}\ C_1\ \texttt{else}\ C_2\ \texttt{fi}\ ;\ C\ \{\Pi' \mid \Sigma'\}}$$

the fresh variable x' is used to record (at least the existence of) a previous value for x. Our fragment here is quantifier-free, but we can still use the same general idea as in the Floyd axiom, as long as we have an overall postcondition and a continuation of the assignment command.

$$\frac{\{x{=}E[x'/x] \wedge A[x'/x]\}\ C\ \{B\}}{\{A\}\ x{:=}E\ ;\ C\ \{B\}}\ x' \text{ fresh}$$

This rule works in standard Hoare logic: the fact that the Floyd axiom expresses the strongest postcondition translates into the soundness and completeness of this rule. All of the rules mentioning fresh variables are obtained in this way from axioms of Separation Logic. This (standard) trick allows us to use a quantifier-free language.

We will not explicitly give the semantics of commands, but just say that we assume Separation Logic's "fault-avoiding" semantics of triples in:

Theorem 1. *All of the operational rules are sound (preserving validity), and all except for* DISPOSE *are complete (preserving invalidity).*

To see the incompleteness of the DISPOSE rule consider:

$$\{x{\mapsto}[] * y{\mapsto}[]\}\ \texttt{dispose}(x)\ ;\ \texttt{empty}\ \{x{\neq}y \mid y{\mapsto}[]\}$$

This is a true triple, but if we apply the DISPOSE and EMPTY rules upwards we will be left with an entailment $y{\mapsto}[] \vdash x{\neq}y \mid y{\mapsto}[]$ that is false. The rule loses the implied information that x and y are unequal from the precondition. Although we can construct artificial examples like this that fool the rule, none of the naturally-occurring examples that we have tried in Smallfoot have suffered from it. The reason, so it seems, is required that inequalities tend to be indicated in boolean conditions in programs, in either while loops or conditionals.

This incompleteness could be dealt with if we were instead to use the backwards-running weakest preconditions of Separation Logic [4]. Unfortunately, there is no existing automatic theorem prover which can deal with the form of these assertions (which use quantification and the separating implication $-\!\!*$). If there were such a prover, we would be eager consumers of it.

Also, there are various hacks to reduce the incompleteness while still reasoning forwards, but actually achieving completeness is nontrivial. So in lieu of neither completeness nor practical problems with the incompleteness, we have opted for the simple solution presented here.

3.2 Rearrangement Rules

The operational rules are not sufficient on their own, because some of them expect their preconditions to be in particular forms. For instance, in

$$\{x{=}y \mid z{\mapsto}[f\colon w] * y{\mapsto}[f\colon z]\}\ x{\to}f{:=}\,y\ ;\ C\ \{\Pi' \mid \Sigma'\}$$

the MUTATE rule cannot fire (be applied upwards), because the precondition has to explicitly have $x{\mapsto}[\rho]$ for some ρ.

Symbolic execution has a separate rearrangement phase, which attempts to put the precondition in the proper form for an operational rule to fire. For instance, in the example just given we can observe that the precondition $x{=}y \mid z{\mapsto}[f\colon w] * y{\mapsto}[f\colon z]$ is equivalent to $x{=}y \mid z{\mapsto}[f\colon w] * x{\mapsto}[f\colon z]$, which is in a form that allows the MUTATE rule to fire.

We use notation for atomic commands that access heap cell E:

$$A(E) ::= E{\to}f{:=}\,F \mid x{:=}\,E{\to}f \mid \texttt{dispose}(E)$$

The basic rearrangement rule simply makes use of equalities to recognize that a dereferencing step is possible.

SWITCH(E)

$$\frac{\{\Pi \mid \Sigma * E{\mapsto}[\rho]\}\ A(E)\,;C\ \{\Pi' \mid \Sigma'\}}{\{\Pi \mid \Sigma * F{\mapsto}[\rho]\}\ A(E)\,;C\ \{\Pi' \mid \Sigma'\}}\ \Pi \mid \Sigma * F{\mapsto}[\rho] \vdash E{=}F$$

For trees and list segments we have rules that expose $\mapsto$ facts by unrolling their inductive definitions, when we have enough information to conclude that

the tree or the list is nonempty.[3] A nonempty tree is one that is not nil.

$$\begin{array}{l}\textsc{Unroll Tree}(E)\\ \dfrac{\{\Pi \mid \Sigma * E\mapsto[l\colon x', r\colon y'] * \mathsf{tree}(x') * \mathsf{tree}(y')\}\ A(E)\ ; C\ \{\Pi' \mid \Sigma'\}}{\{\Pi \mid \Sigma * \mathsf{tree}(F)\}\ A(E)\ ; C\ \{\Pi' \mid \Sigma'\}}\ \dagger\end{array}$$

$$^{\dagger}\text{when } \Pi \mid \Sigma * \mathsf{tree}(F) \vdash F{\neq}\mathsf{nil} \wedge F{=}E \ \text{ and } x', y' \text{ fresh}$$

Here, we have placed the "side condition", which is necessary for the rule to apply, below it, for space reasons. Besides unrolling the tree definition some matching is included using the equality $F{=}E$.

To unroll a list segment we need to know that the beginning and ending points are different, which implies that it is nonempty.

$$\begin{array}{l}\textsc{Unroll List Segment}(E)\\ \dfrac{\{\Pi \mid \Sigma * E\mapsto[n\colon x'] * \mathsf{ls}(x', F')\}\ A(E)\ ; C\ \{\Pi' \mid \Sigma'\}}{\{\Pi \mid \Sigma * \mathsf{ls}(F, F')\}\ A(E)\ ; C\ \{\Pi' \mid \Sigma'\}}\ \dagger\end{array}$$

$$^{\dagger}\text{when } \Pi \mid \Sigma * \mathsf{ls}(F, F') \vdash F{\neq}F' \wedge E{=}F \text{ and } x' \text{ fresh}$$

These rearrangement rules are very deterministic, and are not complete on their own. The reason is that it is possible for an assertion to imply that a cell is allocated, without knowing which $*$-conjunct it necessarily lies in. For example, the assertion $y{\neq}z \mid \mathsf{ls}(x, y) * \mathsf{ls}(x, z)$ contains a "spooky disjunction": it implies that one of the two list segments is nonempty, so that $x{\neq}y \vee x{\neq}z$, but we do not know which. To deal with this in the rearrangement phase we rely on a procedure for exorcising these spooky disjunctions. In essence, $\mathit{exor}(\Pi \mid \Sigma, E)$ is a collection of assertions obtained by doing enough case analysis (adding equalities and inequalities to Π) so that the location of E within a $*$-conjunct is determined. This makes the rearrangement rules complete.

We omit a formal definition of *exor* for space reasons. It is mentioned in the symbolic execution algorithm below, where $\mathit{exor}(g, E)$ is obtained from triple g by applying *exor* to the precondition.

3.3 Symbolic Execution Algorithm

The symbolic execution algorithm works by proof-search using the operational and rearrangement rules. Rearrangement is controlled to ensure termination.

To describe symbolic execution we presume an oracle $\mathit{oracle}(\Pi \mid \Sigma \vdash \Pi' \mid \Sigma')$ for deciding entailments. We also use that we can express consistency of a symbolic heap, and allocatedness, using entailments:

$$\begin{aligned} \mathit{incon}(\Pi \mid \Sigma) &\stackrel{\text{def}}{=} \mathit{oracle}(\Pi \mid \Sigma \vdash \mathsf{nil}{\neq}\mathsf{nil} \mid \mathsf{emp}) \\ \mathit{allocd}(\Pi \mid \Sigma, E) &\stackrel{\text{def}}{=} \mathit{incon}(\Pi \mid \Sigma * E\mapsto[\,]) \text{ and } \mathit{incon}(E{=}\mathsf{nil} \wedge \Pi \mid \Sigma) \end{aligned}$$

We also use $\mathit{pre}(g)$ to denote the precondition in a Hoare triple g. *incon* and *pre* are used to check the precondition for inconsistency in the first step of the symbolic execution algorithm and *allocd* is used int he second-last line.

[3] This is somewhat akin to the "focus" step in shape analysis [11].

Definition 2. *E is active in g if g is of the form*

$$\{\Pi \mid \Sigma\}\ A(E)\,;C\ \{\Pi' \mid \Sigma'\}$$

Algorithm 3 (Symbolic Execution) *Given a triple g, determines whether or not it is provable.*

check(g) =
  if *incon*(*pre*(g)) **return** *"true"*
  if *g matches the conclusion of an operational rule*
    let *p be the premise, or p_1, p_2 the two premises* **in**
    if *rule* EMPTY **return** *oracle*(p)
    if *rule* ASSIGN, MUTATE, NEW, DISPOSE, *or* LOOKUP **return** *check*(p)
    if *rule* CONDITIONAL **return** *check*(p_1) $\wedge$ *check*(p_2)
  elseif *g begins with $A(E)$*
    if SWITCH(E), UNROLL LIST SEGMENT(E), *or* UNROLL TREE(E) *applies*
      let *p be the premise* **in return** *check*(p)
    elseif *allocd*(*pre*(g), E) **return** $\bigwedge\{check(g') \mid g' \in exor(g, E)\}$
    else return *"false"*

Theorem 4. *The Symbolic Execution algorithm terminates, and returns "true" iff there is a proof of the input judgment using the operational and rearrangement rules, where we view each use of an entailment in the symbolic execution rules as a call to the oracle.*

4 Proof Rules for Entailments

The entailment $\Pi \mid \Sigma \vdash \Pi' \mid \Sigma'$ was treated as an oracle in the description of symbolic execution. We now describe a proof theory for entailment.

The rules come in two groups. The first, the normalization rules, get rid of equalities as soon as possible so that the forthcoming rules can be formulated using simple pattern matching (*i.e.*, we can use $E {\mapsto} F$ rather than $E' {\mapsto} F$ plus $E'{=}E$ derivable), make derivable inequalities explicit, perform case analysis using a form of excluded middle, and recognize inconsistency. The second group of rules, the subtraction rules, work by explicating and then removing facts from the right-hand side of an entailment, with the eventual aim of reducing to the axiom $\Pi \mid \mathsf{emp} \vdash \mathsf{true} \mid \mathsf{emp}$.

Before giving the rules, we introduce some notation. We write $op(E)$ as an abbreviation for $E{\mapsto}[\rho]$, $\mathsf{ls}(E, E')$, or $\mathsf{tree}(E)$. The guard $G(op(E))$ is defined by:

$$G(E{\mapsto}[\rho]) \stackrel{\text{def}}{=} \mathsf{true} \qquad G(\mathsf{ls}(E, E')) \stackrel{\text{def}}{=} E{\neq}E' \qquad G(\mathsf{tree}(E)) \stackrel{\text{def}}{=} E{\neq}\mathsf{nil}$$

The proof rules are given in Table 2. Except for $G(op_1(E_1)), G(op_2(E_2)) \in \Pi$, the side-conditions are not needed for soundness, but ensure termination.

Theorem 5 (Soundness and Completeness). *Any provable entailment is valid, and any valid entailment is provable.*

Table 2 Proof System for Entailment

Normalization Rules:

$$\frac{}{\Pi \wedge E{\neq}E \mid \Sigma \vdash \Pi' \mid \Sigma'}$$

$$\frac{\Pi[E/x] \mid \Sigma[E/x] \vdash \Pi'[E/x] \mid \Sigma'[E/x]}{\Pi \wedge x{=}E \mid \Sigma \vdash \Pi' \mid \Sigma'} \qquad \frac{\Pi \mid \Sigma \vdash \Pi' \mid \Sigma'}{\Pi \wedge E{=}E \mid \Sigma \vdash \Pi' \mid \Sigma'}$$

$$\frac{\Pi \wedge G(op(E)) \wedge E{\neq}\mathsf{nil} \mid op(E) * \Sigma \vdash \Pi' \mid \Sigma'}{\Pi \wedge G(op(E)) \mid op(E) * \Sigma \vdash \Pi' \mid \Sigma'} \; E{\neq}\mathsf{nil} \notin \Pi \wedge G(op(E))$$

$$\frac{\Pi \wedge E_1{\neq}E_2 \mid op_1(E_1) * op_2(E_2) * \Sigma \vdash \Pi' \mid \Sigma'}{\Pi \mid op_1(E_1) * op_2(E_2) * \Sigma \vdash \Pi' \mid \Sigma'} \; \begin{array}{l} G(op_1(E_1)), G(op_2(E_2)) \in \Pi \\ E_1{\neq}E_2 \notin \Pi \end{array}$$

$$\frac{\begin{array}{c}\Pi \wedge E_1{=}E_2 \mid \Sigma \vdash \Pi' \mid \Sigma' \\ \Pi \wedge E_1{\neq}E_2 \mid \Sigma \vdash \Pi' \mid \Sigma'\end{array}}{\Pi \mid \Sigma \vdash \Pi' \mid \Sigma'} \; \begin{array}{l} E_1 \not\equiv E_2 \\ E_1{=}E_2, E_1{\neq}E_2 \notin \Pi \\ fv(E_1, E_2) \subseteq fv(\Pi, \Sigma, \Pi', \Sigma') \end{array}$$

$$\frac{\Pi \mid \Sigma \vdash \Pi' \mid \Sigma'}{\Pi \mid \Sigma * \mathsf{tree}(\mathsf{nil}) \vdash \Pi' \mid \Sigma'} \qquad \frac{\Pi \mid \Sigma \vdash \Pi' \mid \Sigma'}{\Pi \mid \Sigma * \mathsf{ls}(E, E) \vdash \Pi' \mid \Sigma'}$$

Subtraction Rules:

$$\frac{}{\Pi \mid \mathsf{emp} \vdash \mathsf{true} \mid \mathsf{emp}} \qquad \frac{\Pi \mid \Sigma \vdash \Pi' \mid \Sigma'}{\Pi \mid \Sigma \vdash \Pi' \wedge E{=}E \mid \Sigma'} \qquad \frac{\Pi \wedge P \mid \Sigma \vdash \Pi' \mid \Sigma'}{\Pi \wedge P \mid \Sigma \vdash \Pi' \wedge P \mid \Sigma'}$$

$$\frac{S \vdash S' \qquad \Pi \mid \Sigma \vdash \Pi' \mid \Sigma'}{\Pi \mid S * \Sigma \vdash \Pi' \mid S' * \Sigma'} \qquad \frac{}{S \vdash S} \qquad \frac{}{E{\mapsto}[\rho, \rho'] \vdash E{\mapsto}[\rho]}$$

$$\frac{\Pi \mid \Sigma \vdash \Pi' \mid \Sigma'}{\Pi \mid \Sigma \vdash \Pi' \mid \mathsf{tree}(\mathsf{nil}) * \Sigma'} \qquad \frac{\Pi \mid \Sigma \vdash \Pi' \mid \Sigma'}{\Pi \mid \Sigma \vdash \Pi' \mid \mathsf{ls}(E, E) * \Sigma'}$$

$$\frac{\Pi \mid E{\mapsto}[l\colon E_1, r\colon E_2, \rho] * \Sigma \vdash \Pi' \mid E{\mapsto}[l\colon E_1, r\colon E_2, \rho] * \mathsf{tree}(E_1) * \mathsf{tree}(E_2) * \Sigma'}{\Pi \mid E{\mapsto}[l\colon E_1, r\colon E_2, \rho] * \Sigma \vdash \Pi' \mid \mathsf{tree}(E) * \Sigma'} \; \dagger$$

$^{\dagger} E{\mapsto}[l\colon E_1, r\colon E_2, \rho] \notin \Sigma'$

$$\frac{\Pi \wedge E_1{\neq}E_3 \mid E_1{\mapsto}[n\colon E_2, \rho] * \Sigma \vdash \Pi' \mid E_1{\mapsto}[n\colon E_2, \rho] * \mathsf{ls}(E_2, E_3) * \Sigma'}{\Pi \wedge E_1{\neq}E_3 \mid E_1{\mapsto}[n\colon E_2, \rho] * \Sigma \vdash \Pi' \mid \mathsf{ls}(E_1, E_3) * \Sigma'} \; \dagger$$

$^{\dagger} E_1{\mapsto}[n\colon E_2, \rho] \notin \Sigma'$

$$\frac{\Pi \mid \mathsf{ls}(E_1, E_2) * \Sigma \vdash \Pi' \mid \mathsf{ls}(E_1, E_2) * \mathsf{ls}(E_2, \mathsf{nil}) * \Sigma'}{\Pi \mid \mathsf{ls}(E_1, E_2) * \Sigma \vdash \Pi' \mid \mathsf{ls}(E_1, \mathsf{nil}) * \Sigma'}$$

$$\frac{\Pi \wedge G(op(E_3)) \mid \mathsf{ls}(E_1, E_2) * op(E_3) * \Sigma \vdash \Pi' \mid \mathsf{ls}(E_1, E_2) * \mathsf{ls}(E_2, E_3) * \Sigma'}{\Pi \wedge G(op(E_3)) \mid \mathsf{ls}(E_1, E_2) * op(E_3) * \Sigma \vdash \Pi' \mid \mathsf{ls}(E_1, E_3) * \Sigma'}$$

The side-conditions are sufficient to ensure that progress is made when applying rules upwards. Decidability then follows using the naive proof procedure which tries all possibilities, backtracking when necessary.

Theorem 6 (Decidability). *Entailment is decidable.*

It is possible, however, to do much better than the naive procedure. For example, one narrowing of the search space is a phase distinction between normalization and subtraction rules: Any subtraction rule can be commuted above any normalization rule. Further commutations are possible for special classes of assertion, and these are used in Smallfoot.

This system's proof rules can be viewed as coming from certain implications, and are arranged as rules just to avoid the explicit use of the cut rule in proof search. For instance, the fourth normalization rule comes from the implications:

$$E \mapsto [] \rightarrow E \neq \mathsf{nil} \qquad\qquad E_1 \neq E_2 \wedge \mathsf{ls}(E_1, E_2) \rightarrow E_1 \neq \mathsf{nil}$$

the fifth from the implications:

$$E_1 \mapsto [\rho_1] * E_2 \mapsto [\rho_2] \rightarrow E_1 \neq E_2 \qquad\qquad E_2 \neq \mathsf{nil} \wedge E_1 \mapsto [\rho] * \mathsf{tree}(E_2) \rightarrow E_1 \neq E_2$$

$$E_2 \neq E_3 \wedge E_1 \mapsto [\rho] * \mathsf{ls}(E_2, E_3) \rightarrow E_1 \neq E_2$$

$$E_1 \neq \mathsf{nil} \wedge E_2 \neq \mathsf{nil} \wedge \mathsf{tree}(E_1) * \mathsf{tree}(E_2) \rightarrow E_1 \neq E_2$$

$$E_1 \neq \mathsf{nil} \wedge E_2 \neq E_3 \wedge \mathsf{tree}(E_1) * \mathsf{ls}(E_2, E_3) \rightarrow E_1 \neq E_2$$

$$E_1 \neq E_3 \wedge E_2 \neq E_4 \wedge \mathsf{ls}(E_1, E_3) * \mathsf{ls}(E_2, E_4) \rightarrow E_1 \neq E_2$$

and the last two from the implications:

$$\mathsf{tree}(\mathsf{nil}) \rightarrow \mathsf{emp} \qquad\qquad \mathsf{ls}(E, E) \rightarrow \mathsf{emp}$$

For the inductive predicates, these implications are consequences of unrolling the inductive definition in the metatheory. But note that we do not unroll predicates, instead case analysis via excluded middle takes one judgment to several.

Likewise, the subtraction rules for the inductive predicates are obtained from the implications:

$$\mathsf{emp} \rightarrow \mathsf{tree}(\mathsf{nil}) \qquad\qquad E \mapsto [l\colon E_1, r\colon E_2, \rho] * \mathsf{tree}(E_1) * \mathsf{tree}(E_2) \rightarrow \mathsf{tree}(E)$$

$$\mathsf{emp} \rightarrow \mathsf{ls}(E, E) \qquad\qquad E_1 \neq E_3 \wedge E_1 \mapsto [n\colon E_2, \rho] * \mathsf{ls}(E_2, E_3) \rightarrow \mathsf{ls}(E_1, E_3)$$

$$\mathsf{ls}(E_1, E_2) * \mathsf{ls}(E_2, \mathsf{nil}) \rightarrow \mathsf{ls}(E_1, \mathsf{nil})$$

$$\mathsf{ls}(E_1, E_2) * \mathsf{ls}(E_2, E_3) * E_3 \mapsto [\rho] \rightarrow \mathsf{ls}(E_1, E_3) * E_3 \mapsto [\rho]$$

$$E_3 \neq \mathsf{nil} \wedge \mathsf{ls}(E_1, E_2) * \mathsf{ls}(E_2, E_3) * \mathsf{tree}(E_3) \rightarrow \mathsf{ls}(E_1, E_3) * \mathsf{tree}(E_3)$$

$$E_3 \neq E_4 \wedge \mathsf{ls}(E_1, E_2) * \mathsf{ls}(E_2, E_3) * \mathsf{ls}(E_3, E_4) \rightarrow \mathsf{ls}(E_1, E_3) * \mathsf{ls}(E_3, E_4)$$

The first four are straightforward, while the last four express properties whose verification of soundness would use inductive proofs in the metatheory. The resulting rules do not, however, require a search for inductive premises. In essence,

what we generally do is, for each considered inductive predicate, add a collection of rules that are consequences of induction, but that can be formulated in a way that preserves the proof theory's terminating nature.

In the last subtraction rule, the $G(op(E_3)) \wedge op(E_3)$ part of the left-hand side ensures that E_3 does not occur within the segments from E_1 to E_2 or from E_2 to E_3. This is necessary for appending list segments, since they are required to be acyclic.

Here is an example proof, of the entailment mentioned in the Introduction:

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{t{\neq}\mathsf{nil} \mid \mathsf{emp} \vdash \mathsf{emp}}{t{\neq}\mathsf{nil} \mid \mathsf{ls}(y,\mathsf{nil}) \vdash \mathsf{ls}(y,\mathsf{nil})}}{t{\neq}\mathsf{nil} \mid t{\mapsto}[n\colon y] * \mathsf{ls}(y,\mathsf{nil}) \vdash t{\mapsto}[n\colon y] * \mathsf{ls}(y,\mathsf{nil})}}{t{\neq}\mathsf{nil} \mid t{\mapsto}[n\colon y] * \mathsf{ls}(y,\mathsf{nil}) \vdash \mathsf{ls}(t,\mathsf{nil})}}{t{\neq}\mathsf{nil} \mid \mathsf{ls}(x,t) * t{\mapsto}[n\colon y] * \mathsf{ls}(y,\mathsf{nil}) \vdash \mathsf{ls}(x,\mathsf{nil})}}{\mathsf{ls}(x,t) * t{\mapsto}[n\colon y] * \mathsf{ls}(y,\mathsf{nil}) \vdash \mathsf{ls}(x,\mathsf{nil})}$$

Going upwards, this applies the normalization rule which introduces $t{\neq}\mathsf{nil}$, then the subtraction rule for nil-terminated list segments, the subtraction rule for nonempty list segments, and finally $*$-Introduction (the basic subtraction rule for $*$, which appears fourth) twice.

5 Incomplete Proofs and Frame Axioms

Typically, at a call site to a procedure the symbolic heap will be larger than that required by the procedure's precondition. This is the case in the `DispTree` example where, for example, the symbolic heap at one of the recursive call sites is $p{\mapsto}[l\colon i, r\colon j] * \mathsf{tree}(i) * \mathsf{tree}(j)$, where that expected by the procedure specification of `DispTree`(i) is just $\mathsf{tree}(i)$. We show how to use the proof theory from the previous section to infer frame axioms.

In more detail the (spatial) part of the problem is,

> *Given*: two symbolic heaps, $\Pi \mid \Sigma$ (the heap at the call site), and $\Pi_1 \mid \Sigma_1$ (the procedure precondition)
> *To Find*: a spatial predicate Σ_F, the "spatial frame axiom", satisfying the entailment $\Pi \mid \Sigma \vdash \Pi_1 \mid \Sigma_1 * \Sigma_F$.

Our strategy is to search for a proof of the judgment $\Pi \mid \Sigma \vdash \Pi_1 \mid \Sigma_1$, and if this search, going upwards, halts at $\Pi' \mid \Sigma_F \vdash \mathsf{true} \mid \mathsf{emp}$ then Σ_F is a sound choice as a frame axiom. We give a few examples to show how this mechanism works.

First, and most trivially, let us consider the `DispTree` example:

> Assertion at Call Site : $p{\mapsto}[l\colon i, r\colon j] * \mathsf{tree}(i) * \mathsf{tree}(j)$
> Procedure Precondition : $\mathsf{tree}(i)$

Then an instance of $*$-INTRODUCTION

$$\frac{p\mapsto[l\colon i, r\colon j] * \mathsf{tree}(j) \vdash \mathsf{emp}}{p\mapsto[l\colon i, r\colon j] * \mathsf{tree}(i) * \mathsf{tree}(j) \vdash \mathsf{tree}(i)}$$

immediately furnishes the correct frame axiom: $p\mapsto[l\colon i, r\colon j] * \mathsf{tree}(j)$.

For an example that requires a little bit more logic, consider:

Assertion at Call Site : $x\mapsto[\,] * y\mapsto[\,]$
Procedure Precondition : $x\neq y \mid x\mapsto[\,]$

$$\cfrac{\cfrac{\cfrac{x\neq y \mid y\mapsto[\,] \vdash \mathsf{emp}}{x\neq y \mid y\mapsto[\,] \vdash x\neq y \mid \mathsf{emp}}}{x\neq y \mid x\mapsto[\,] * y\mapsto[\,] \vdash x\neq y \mid x\mapsto[\,]}}{x\mapsto[\,] * y\mapsto[\,] \vdash x\neq y \mid x\mapsto[\,]}$$

Here, the inequality $x\neq y$ is added to the left-hand side in the normalization phase, and then it is removed from the right-hand side in the subtraction phase.

On the other hand, consider what happens in a wrong example:

Assertion at Call Site : $x\mapsto[\,] * y\mapsto[\,]$
Procedure Precondition : $x=y \mid x\mapsto[\,]$

$$\cfrac{\cfrac{\cfrac{??}{x\neq y \mid y\mapsto[\,] \vdash x=y \mid \mathsf{emp}}}{x\neq y \mid x\mapsto[\,] * y\mapsto[\,] \vdash x=y \mid x\mapsto[\,]}}{x\mapsto[\,] * y\mapsto[\,] \vdash x=y \mid x\mapsto[\,]}$$

In this case we get stuck at an earlier point because we cannot remove the equality $x=y$ from the right-hand side in the subtraction phase. To correctly get a frame axiom we have to obtain true in the pure part of the right-hand side; we do not do so in this case, and we rightly do not find a frame axiom.

The proof-theoretic justification for this method is the following.

Theorem 7. *Suppose that we have an incomplete proof (a proof that doesn't use axioms):*

$$\begin{array}{c}[\Pi' \mid \Sigma_F \vdash \mathsf{true} \mid \mathsf{emp}]\\ \vdots\\ \Pi \mid \Sigma \vdash \Pi_1 \mid \Sigma_1\end{array}$$

Then there is a complete proof (without premises, using an axiomatic rule at the top) of:

$$\Pi \mid \Sigma \vdash \Pi_1 \mid \Sigma_1 * \Sigma_F.$$

This justifies an extension to the symbolic execution algorithm. In brief, we extend the syntax of loop-free triples with a **jsr** instruction

$$C ::= \cdots \mid [\Pi \mid \Sigma]\ \mathbf{jsr}\ [\Pi' \mid \Sigma']\,; C \quad \text{jump to subroutine}$$

annotated with a precondition and a postcondition. In Smallfoot this is is generated when an annotated program is chopped into straightline Hoare triples. The appropriate operational rule is:

$$\frac{\Pi \mid \Sigma \vdash \Pi_1 \wedge \Pi \mid \Sigma_1 * \Sigma_F \qquad \{\Pi_2 \wedge \Pi \mid \Sigma_2 * \Sigma_F\}\ C\ \{\Pi' \mid \Sigma'\}}{\{\Pi \mid \Sigma\}\ [\Pi_1 \mid \Sigma_1]\ \mathbf{jsr}\ [\Pi_2 \mid \Sigma_2]\,;C\ \{\Pi' \mid \Sigma'\}}$$

When we encounter a **jsr** command during symbolic execution we run the proof theory from the previous section upwards with goal $\Pi \mid \Sigma \vdash \Pi_1 \mid \Sigma_1$. If it terminates with $\Pi' \mid \Sigma_F \vdash \mathsf{true} \mid \mathsf{emp}$ then we tack Σ_F onto the postcondition Σ_2, and we continue execution with C. Else we report an error.

The description here is simplified. Theorem 7 only considers incomplete proofs with single assumptions, but it is possible to generalize the treatment of frame inference to proofs with multiple assumptions (which leads to several frames being checked in symbolic execution). Also, we have only discussed the spatial part of the frame, neglecting modifies clauses for stack variables. A pure frame must also be discovered, but that is comparatively easy.

6 Conclusion

We believe that symbolic execution with Separation Logic has some promise for modular verification and analysis. PALE is (purposely) unsound in its treatment of frame axioms for procedures [7], the "modular soundness" of ESC is subtle but probably not definitive [6], interprocedural shape analysis is just beginning to become modular [10], and existing symbolic execution techniques such as [5] are not modular in this sense. But there is much more to be done on our part, even apart from investigating fixed-point convergence and widening. We would like to have a general scheme of inductive definitions rather than using hardwired predicates. (We are not just asking for semantically well-defined recursive predicates, *e.g.*, as developed in [12], but would want a, hopefully terminating, proof theory.) It would be desirable to incorporate our proof theory with other decision procedures, such as based on monadic second-order logic or shape analysis or Pressburger arithmetic. And so on. So clearly, this is but a start.

References

1. J. Berdine, C. Calcagno, and P. O'Hearn. A decidable fragment of separation logic. Proceedings of FSTTCS, LNCS 3328, Chennai, December, 2004.
2. J. Berdine C. Calcagno and P.W. O'Hearn. Smallfoot: A tool for Checking Separation Logic footprint specifications. In preparation, 2005.
3. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. 4th ACM Symposium on Principles of Programming Languages. pages 238–252, 1977.
4. S. Isthiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 36–49, London, January 2001.

5. S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. In Hubert Garavel and John Hatcliff, editors, *Proceedings of TACAS*, volume 2619 of *LNCS*, pages 553–568. Springer, 2003.
6. K. R. M. Leino and G. Nelson. Data abstraction and information hiding. *ACM Trans. Program. Lang. Syst.*, 24(5):491–553, 2002.
7. A. Moller and M. Schwartzbach. The pointer assertion logic engine. Proceedings of PLDI, 221-231, 2001.
8. P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Proceedings of 15th Annual Conference of the European Association for Computer Science Logic*, LNCS, pages 1–19. Springer-Verlag, 2001.
9. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. Invited Paper, *Proceedings of the 17th IEEE Symposium on Logic in Computer Science*, pages 55-74, 2002.
10. N. Rinetzky, J. Bauer, T. Reps, S. Sagiv, and R. Wilhelm. A semantics for procedure local heaps and its abstractions. *32nd POPL*, pp296–309, 2005.
11. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.
12. É.-J. Sims. Extending separation logic with fixpoints and postponed substitution. In *AMAST Proceedings*, pages 475–490, 2004. Springer LNCS 3116.

Heap-Abstraction for an Object-Oriented Calculus with Thread Classes*

Erika Ábrahám[1] and Andreas Grüner[2] and Martin Steffen[2]

[1] Albert-Ludwigs-University Freiburg, Germany
[2] Christian-Albrechts-University Kiel, Germany

Abstract. From an observational point of view, considering classes as part of a component makes instantiation a possible interaction between component and environment or observer. For thread classes it means that a component may create external activity, which influences what can be observed. The fact that cross-border instantiation is possible requires that the *connectivity* of the objects needs to be incorporated into the semantics. We extend our prior work not only by adding thread classes, but also in that thread names may be *communicated,* which means that the semantics needs to account explicitly for the possible acquaintance of objects with threads.
This paper formalizes an open semantics for a calculus featuring thread classes, where the environment, consisting in particular of an overapproximation of the heap topology, is abstractly represented. We show basic soundness results of the abstraction.

Keywords: class-based oo languages, thread-based concurrency, open systems, formal semantics, heap abstraction, observable behavior

1 Introduction

An *open* system is a program fragment or component interacting with its environment or context. In a message-passing setting, the behavior of the component can be understood to consist of message traces at the interface, i.e., of sequences of component-environment interaction. Even if the environment is absent, it must be assumed that the component together with the (abstracted) environment gives a well-formed program adhering to the syntactical and the context-sensitive restrictions of the language at hand. Technically, for an exact representation of the interface behavior, the semantics of the open program needs to be formulated under *assumptions* about the environment, capturing those restrictions. The resulting assumption-commitment framework gives insight to the semantical nature of the language. Furthermore, an independent characterization of possible interface behavior with environment and component

* Part of this work has been financially supported by the NWO/DFG project Mobi-J (RO 1122/9-4).

abstracted can be seen as a trace logic under the most general assumptions, namely conformance to the inherent restrictions of the very language and its semantics.

With these goals in mind, this paper deals primarily with the following three features, which correspond to those of modern class-based object-oriented languages like *Java* [10] or $C^{\#}$ [8] and which are notoriously hard to capture:

- *types and classes:* the languages are statically typed, and only well-typed programs are considered. For class-based languages, complications arise as classes play the role of types and additionally act as *generators* of objects.
- *concurrency:* the mentioned languages feature concurrency based on *threads* (as opposed to processes or active objects).
- *references:* each object carries a unique *identity.* New objects are dynamically allocated on the heap as *instances of classes.*

We investigate the issues in a class-based, multi-threaded calculus with thread classes. The interface behavior is phrased in an assumption-commitment framework and based on three orthogonal abstractions:

- a static abstraction, i.e., the type system;
- an abstraction of the stacks of recursive method invocations, representing the recursive and reentrant nature of method calls in a multi-threaded setting;
- finally an abstraction of the *heap topology,* approximating potential connectivity of objects and threads. The heap topology is dynamic in that new objects may be created and tree structured in that previously separate groups of objects may merge.

In [3,4] we showed that the last point, namely the need to represent the heap topology, is a direct consequence of considering *classes* as a language concept. Their foremost role in object-oriented languages is to act as *"generators of state"*. With *thread classes,* there is also a mechanism for *"generating new activity"*, i.e., for creating new threads. This extension makes cross-border activity generation a possible component-environment interaction, i.e., the component may create threads in the environment and vice versa.

Thus, the technical contribution of this paper is threefold. We extend the class-based calculus and its semantics of [3,4] to include *thread classes* and furthermore allow the communication of thread names. This requires to consider cross-border *activity* generation as well as to incorporate the connectivity of objects *and* threads. Secondly, we characterize the potential traces of *any* component in an assumption-commitment framework in a novel derivation system, where the branching nature of the heap abstraction —connected groups of objects can merge by communication— is reflected in the branching structure of the derivation system. Finally, we show the soundness of the mentioned abstractions.

Overview The paper is organized as follows. Section 2 contains syntax and operational semantics of the calculus we use, formalizing the notion of thread

classes. Section 3 contains an independent characterization of the observable behavior of an open system and the soundness results of the abstractions. Section 4 concludes with related and future work. For a full account of the operational semantics and the type system, we refer to the technical report [5].

2 A multi-threaded calculus with thread classes

Next we present the calculus, starting with the syntax. It is based on the multi-threaded object calculus, similar to the one presented in [9] and in particular [11]. Compared to our previous work for instance in [2], we added thread classes as generators of activity.

2.1 Syntax

The abstract syntax is given in Table 1. A program is given by a collection of classes where a class $c[\![O]\!]$ carries a name c and defines the implementation of its methods and fields. *Thread classes,* written $c_t\langle\!\langle t_a\rangle\!\rangle$, are known under the name c_t and carry the code in t_a. For names, we will generally use o and its syntactic variants as names for objects, c for classes (in particular c_t for thread classes), and n when being unspecific, for instance in Table 1.

An object $o[c, F]$ stores the current value of the fields or instance variables and keeps a reference to the class it instantiates. A method $\varsigma(n{:}c).\lambda(x_1{:}T_1, \ldots, x_k{:}T_k).t$ provides the method body abstracted over the ς-bound "self" parameter and the formal parameters of the method [1]. Besides named objects and classes, the dynamic configuration of a program contains threads $n\langle t\rangle$ as active entities.

A thread is basically either a value or a sequence of expressions, notably method calls (written $v.l(\vec{v})$), the creation of new objects $new\, c$ where c is a class name, and *thread instantiation* written as $spawn\, c_t(\vec{v})$.

Furthermore we will use f for instance variables or fields, we use $f = v$ for field variable declaration, field access is written as $x.f$, and field update as $x.f := v$.

The available types include *thread* as the type of threads. Furthermore, objects are typed by the name of their class. As auxiliary types we have $T_1 \times \ldots \times T_k \to T$ as the type of methods as well as for thread classes (in which case the result type T equals *thread*), and furthermore $[l_1{:}U_1, \ldots, l_k{:}U_k]$ as the type or interface of unnamed objects, and $[\![l_1{:}U_1, \ldots, l_k{:}U_k]\!]$ as the type for classes.

2.2 Operational semantics

For the operational semantics, we concentrate on the interface behavior. For want of space, we omit the (straightforward) definitions of the component-internal steps, for instance-internal method calls or internal thread creation. For the definition of the semantics, we refer to [5].

The external steps define the interaction of the component with the environment. In particular, the semantics is defined in reference to assumption and

$C ::= \mathbf{0} \mid C \parallel C \mid \nu(n{:}T).C \mid n[\![O]\!] \mid n[n, F] \mid n\langle t\rangle \mid n\langle\!\langle t_a\rangle\!\rangle$	program
$O ::= F, M$	object
$M ::= l = m, \ldots, l = m$	method suite
$F ::= l = f, \ldots, l = f$	fields
$m ::= \varsigma(n{:}T).\lambda(x{:}T, \ldots, x{:}T).t$	method
$f ::= \varsigma(n{:}T).\lambda().v \mid \varsigma(n{:}T).\lambda().\,stop$	field
$t_a ::= \lambda(x{:}T, \ldots, x{:}T).t$	thread abstraction
$t ::= v \mid stop \mid let\, x{:}T = e\, in\, t$	thread
$e ::= t \mid \mathsf{if}\ v = v\ \mathsf{then}\ e\ \mathsf{else}\ e$	expr.
$\mid\ v.l(v, \ldots, v) \mid v.l := v \mid currentthread$	
$\mid\ new\, n \mid spawn\, n(v, \ldots, v)$	
$v ::= x \mid n$	values

Table 1. Abstract syntax

commitment contexts. The static part of the contexts corresponds to the static type system (we again refer to [5] for the full definition) and takes care that, e.g., only well-typed values are received from the environment. The contexts, however, need to contain also a *dynamic* part dealing with the potential *connectivity* of objects and thread names and which corresponds to an abstraction of the heap of the program.

A component exchanges information with the environment via *calls, returns,* and *spawn* actions (cf. Table 2). In the call and return labels, the mentioned n is the active thread that issues the call or returns from the call. In the thread instantiation label, n is the name of the new thread; the thread which spawned the new thread is *not* part of the label.[3] Furthermore note that there are no separate external labels for object instantiation: Externally instantiated objects are created only at the point when they are actually accessed for the first time, which we call "*lazy instantiation*". Given a label $\nu(\Phi).\gamma'$ where Φ is a name context, i.e., a sequence of single $\nu(n{:}T)$ bindings and where γ' does not contain any binders, we call γ' the *core* of the label. Given a label γ, we refer with $\lfloor\gamma\rfloor$ to its core. Analogously for send and receive labels.

$\gamma ::= n\langle call\ o.l(\vec{v})\rangle \mid n\langle return(v)\rangle \mid \langle spawn\, n\ of\ c(\vec{v})\rangle \mid \nu(n{:}T).\gamma$	basic labels
$a ::= \gamma? \mid \gamma!$	receive and send

Table 2. Labels

[3] Of course it might be mentioned in the arguments.

2.2.1 Connectivity contexts In the presence of cross-border instantiation, the semantics must contain a representation of the connectivity, which will be formalized by a relation on the names of the calculus and which can be seen as an abstraction of the program's heap; for the exact definition, see Equation (2) and (3) below. The external semantics is formalized as labeled transitions between judgments of the form

$$\Delta, \Sigma; E_\Delta \vdash C : \Theta, \Sigma; E_\Theta \ , \tag{1}$$

where $\Delta, \Sigma; E_\Delta$ are the *assumptions* about the environment of the component C and $\Theta, \Sigma; E_\Theta$ the *commitments*. The assumptions consist of a part Δ, Σ concerning the existence (plus static typing information) of *named entities* in the environment. The semantics maintains as invariant that the assumption and commitment contexts are disjoint concerning object and class names, whereas a thread name occurs as assumption iff. it is mentioned in the commitments. By convention, the contexts Σ (and their alphabetic variants) contain exactly all bindings for thread names. This means, as invariant we maintain for all judgments $\Delta, \Sigma; E_\Delta \vdash C : \Theta, \Sigma; E_\Theta$ that Δ, Σ, and Θ are pairwise disjoint.

The semantics must book-keep which objects of the environment have been told which identities. This means it must take into account the *relation* of objects from the assumption context Δ amongst each other, and the knowledge of objects from Δ about thread names and names exported by the component, i.e., those from Θ. In analogy to the name contexts Δ and Θ, E_Δ (the connectivity context) expresses assumptions about the environment, and E_Θ commitments of the component:

$$E_\Delta \subseteq \Delta \times (\Delta + \Sigma + \Theta) \ . \tag{2}$$

and dually $E_\Theta \subseteq \Theta \times (\Theta + \Sigma + \Delta)$. Since in the language we allow the sending of thread names, we must include pairs from $\Delta \times \Sigma$ (resp. $\Theta \times \Sigma$) into the connectivity. We write $o \hookrightarrow n$ ("o may know n") for pairs from the relations E_Δ, resp. E_Θ. Without full information about the complete system, the component must make worst-case assumptions concerning the proliferation of knowledge, which are represented as the *reflexive, transitive,* and *symmetric* closure of the $\hookrightarrow$-pairs of *objects from* Δ. Given Δ, Θ, and E_Δ, we write $\leftrightharpoons$ for this closure, i.e.,

$$\leftrightharpoons \ \triangleq \ (\hookrightarrow\downarrow_\Delta \cup \hookleftarrow\downarrow_\Delta)^* \subseteq \Delta \times \Delta \ , \tag{3}$$

where $\hookrightarrow\downarrow_\Delta$ is the projection of $\hookrightarrow$ to Δ. We also need the union $\leftrightharpoons \cup \leftrightharpoons ; \hookrightarrow \subseteq \Delta \times (\Delta + \Sigma + \Theta)$, where the semicolon denotes relational composition. We write $\leftrightharpoons\!\hookrightarrow$ for that union. As judgment, we use $\Delta, \Sigma; E_\Delta \vdash o_1 \leftrightharpoons o_2 : \Theta, \Sigma$, resp. $\Delta, \Sigma; E_\Delta \vdash o \leftrightharpoons\!\hookrightarrow n : \Theta, \Sigma$. For Θ, Σ, E_Θ, and Δ, Σ, the definitions are applied dually.

The relation $\leftrightharpoons$ partitions the objects from Δ (resp. Θ) into equivalence classes. We call a set of object names from Δ (or dually from Θ) such that for all objects o_1 and o_2 from that set, $\Delta, \Sigma; E_\Delta \vdash o_1 \leftrightharpoons o_2 : \Theta, \Sigma$, a *clique*, and if we speak of *the* clique of an object we mean the equivalence class.

As for the relationship of communicated values, incoming and outgoing communication play dual roles: E_Θ over-approximates the actual connectivity of the

component and is updated in incoming communication, while the assumption context E_Δ is consulted to exclude impossible combinations of incoming values. Incoming new names, exchanged boundedly, however, update both commitments and assumptions.

Remark 1 (Initial clique). Note that a thread can be instantiated *without* connection to any object/clique and indeed the initial thread starts with static code, i.e., without reference to any object. For appropriately dealing with the connectivity in those cases, we need a syntactical representation for the clique of objects, the thread n starts in; we use the symbol $\odot_n$ as n's *initial clique.*

Concerning $\odot_n$, the semantics maintains as invariant that a thread name n occurs in the context Σ for thread names, iff. $\odot_n$ occurs in either Δ or Θ, the contexts containing the objects (plus class definitions). This means, besides being relevant for connectivity information, $\odot_n$ contains also the information whether the thread started its life in the environment or in the component.

2.2.2 Augmentation To formulate the external communication properly, we need to introduce a few augmentations. We extend the syntax by two additional expressions

$$o_1 \ \mathit{blocks\ for}\ o_2 \quad \text{and} \quad o_2\, \mathit{returns\ to}\, o_1\, v\ .$$

The first one denotes a method body in o_1 waiting for a return from o_2, and dually the second expression returns v from o_2 to o_1.

Furthermore, we augment the syntax of the method definitions accordingly, such that each method call and each spawn step is preceded by an annotation of the caller; i.e., instead of $\varsigma(\mathit{self}{:}c).\lambda(\vec{x}{:}\vec{T}).(\ldots x.l(\vec{y})\ldots \mathit{spawn}\, c_t(\vec{z})\ldots)$ we write

$$\varsigma(\mathit{self}{:}c).\lambda(\vec{x}{:}\vec{T}).(\ldots \mathit{self}\, x.l(\vec{y})\ldots \mathit{self\ spawn}\, c_t(\vec{z})\ldots)\ .$$

We need to augment the threads such that every thread n carries at the beginning the identity $\odot_n$ of its initial clique. The program starts with one single initial thread. If the thread starts within the component, the contexts of the initial configuration $\Delta_0 \vdash C : \Theta_0$ asserts $\Theta_0 \vdash \odot$. Otherwise, $\Delta_0 \vdash \odot$. As in the augmentation for methods, the code in the thread classes must be augmented in such a way, that for method calls the initial clique of the thread is mentioned in front of the call. I.e., after instantiation, the call looks as follows: $n\langle \ldots \odot_n\, x.l(\vec{v})\ldots\rangle$. The static code of each thread class is augmented into

$$c_t\langle\!\langle \lambda(\vec{x}{:}\vec{T}).(\ldots \odot\, x.l(\vec{v})\ldots)\rangle\!\rangle$$

for each mentioned call. When the thread is instantiated, $\odot$ is replaced by $\odot_n$ where n is the identity of the new thread. Given the above thread class, we denote by $c_t(\vec{v})$ the replacement $t[\odot_n, \vec{v}/\odot, \vec{x}]$, when t is the body of the thread class definition. The initial thread, which is not instantiated from a thread class but given directly (in case the activity starts in the component) starts with $\odot_n$ as augmentation, if the initial thread is named n. If the component is renamed

by α-conversion, n and $\odot_n$ are renamed simultaneously. The steps of the internal semantics must be adapted accordingly. We also omit the typing rules for the augmentation, as they are straightforward.

2.2.3 Use and change of contexts The operational semantics is formulated as transitions between typed judgments

$$\Delta, \Sigma; E_\Delta \vdash C : \Theta, \Sigma; E_\Theta \xrightarrow{a} \acute{\Delta}, \acute{\Sigma}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}, \acute{\Sigma}; \acute{E}_\Theta \ .$$

The assumption context $\Delta, \Sigma; E_\Delta$ can be seen as an abstraction of the (not-present) environment; more precisely, it represents the potential behavior of all possible environments.

Notation 1 *To facilitate the following definitions notationally, we will make use of the following conventions. We abbreviate the triple of name contexts Δ, Σ, Θ as Φ, and the context $\Delta, \Sigma, \Theta, E_\Delta, E_\Theta$ combining assumptions and commitments Ξ. Furthermore we understand $\acute{\Delta}, \acute{\Sigma}, \acute{\Theta}$ as $\acute{\Phi}$, and $\acute{\Xi}$ as consisting of $\acute{\Delta}, \acute{\Sigma}, \acute{\Theta}, \acute{E}_\Delta, \acute{E}_\Theta$, etc.*

The check whether the current assumptions are met in an incoming communication step is given in Definition 1.

Definition 1 (Connectivity check). *An incoming core label a with sender o_s and receiver o_r is* well-connected *wrt. an assumption-commitment context $\acute{\Xi}$ (written $\acute{\Xi} \vdash o_s \xrightarrow{a} o_r :ok$) if:*

$$\acute{\Delta}, \acute{\Sigma}; \acute{E}_\Delta \vdash o_s \leftrightharpoons\hookrightarrow fn(a) : \acute{\Theta}, \acute{\Sigma} \ . \tag{4}$$

Note that in case of an incoming *call* label, $fn(a)$ includes the receiver o_r and the thread name.

Besides *checking* whether the connectivity assumptions are met before a transition, the contexts are *updated* by a step, reflecting the change of knowledge.

Definition 2 (Name context update: Φ+a). *The* update *$\acute{\Phi}$ of an assumption-commitment context Φ wrt. an incoming label $a = \nu(\Phi')\lfloor a \rfloor$ is defined as follows.*

1. *$\acute{\Theta} = \Theta + \Theta'$. In case of a spawn-label $\acute{\Theta} = \Theta + \Theta', \odot_n$, where n is the name of the spawned thread.*
2. *$\acute{\Delta} = \Delta + \odot_{\Sigma'}, \Delta'$. In case of a spawn label, $\odot_{\Sigma' \setminus n}$ is used instead of $\odot_{\Sigma'}$, where n is the name of the spawned thread.*
3. *$\acute{\Sigma} = \Sigma + \Sigma'$.*

We write $\Phi + a$ for the update. The update for outgoing communication is defined dually in the sense that $\odot_n$ of a spawn label is added to Δ instead of Θ. Likewise, the $\odot_{\Sigma'}$ (resp. $\odot_{\Sigma' \setminus n}$) are added to Θ, instead of Δ. (The notation $\odot_{\Sigma'}$ abbreviates $\odot_n$ for all thread identities from Σ').

Definition 3 (Connectivity context update). *The update* $(\acute{E}_\Delta, \acute{E}_\Theta)$ *of an assumption-commitment context* (E_Δ, E_Θ) *wrt. an incoming label* $a = \nu(\Phi')\lfloor a \rfloor?$ *with sender* o_s *and receiver* o_r *is defined as follows.*

1. $\acute{E}_\Theta = E_\Theta + o_r \hookrightarrow fn(\lfloor a \rfloor)$.
2. $\acute{E}_\Delta = E_\Delta + o_s \hookrightarrow \Phi', \odot_{\Sigma'}$. *In case of a spawn label,* $\odot_{\Sigma' \setminus n}$ *is used instead of* $\odot_{\Sigma'}$*, where* n *is the name of the spawned thread.*

We write $(E_\Delta, E_\Theta) + o_s \xrightarrow{a} o_r$ *for the update.*

Combining Definitions 2 and 3, we write $\Xi + o_s \xrightarrow{a} o_r$ when updating the name and the connectivity at the same time.

Besides Definition 1, which checks whether the connectivity assumptions are met for the label at hand, we must additionally check the *static* assumptions, i.e., whether the transmitted values are of the correct types. In slight abuse of notation, we write $\Delta, \Sigma, \Theta \vdash o_s \xrightarrow{a} o_r : T$ for that check, where T is type of the expression in the program that gives rise to the label. We omit the exact definition here which can be found in [5]. We combine the connectivity check of Definition 1 and the type check notationally into one single judgment $\Xi \vdash o_s \xrightarrow{a} o_r : T$.

2.2.4 Operational rules

With all the ancillary definitions at hand, we can define the operational rules of the semantics (cf. Table 3).

The three CallI-rules deal with incoming calls. For all three cases, the contexts are *updated* to $\acute{\Xi}$ to include the information concerning new objects, threads, and connectivity transmitted in that step. Furthermore, it is *checked* whether the label statically type-checks and that the step is possible according to the (updated) connectivity assumptions $\acute{\Xi}$. Remember that the update from Ξ to $\acute{\Xi}$ includes guessing of connectivity, i.e., an element of non-determinism, when the sender of the communication is unknown to the component.

The three rules for incoming calls deal with three different situations as to when an incoming call may happen: A reentrant call[4], a call of thread where the thread name is already known in the component, and a call of a thread which is new to the component.

To deal with component entities (threads and objects) that are being created during the call $C(\Theta', \Sigma')$ stands for $C(\Theta') \parallel C(\Sigma')$, were $C(\Theta')$ are the lazily instantiated objects mentioned in Θ'. Furthermore, for each thread name n' in Σ', a new component $n'\langle stop \rangle$ is included, written as $C(\Sigma')$.

The treatment of the connectivity contexts is uniform in all three cases, only the identity of the sender is different.

For reentrant method calls (cf. rule CallI$_1$), the thread is blocked, i.e., it has left the component previously via an outgoing call. The object that had been the target of the call is remembered as part of the augmented block syntax. In the rule it is referred to as o_s, as it represents the sender's clique of the current incoming call.

[4] Reentrant on the level of the component, not on the level of a single object.

Rule $\textsc{CallI}_2$ treats a non-reentrancy situation, where the thread name is already known in the component nonetheless. As a consequence, the component contains the entity $n\langle stop\rangle$. Unlike in rule $\textsc{CallI}_1$, the program code contains no indication as to the origin of the call. Since the thread n must have crossed the border before, the marker for its initial clique $\odot_n$ must be contained in either Δ or in Θ. The premise $\Delta \vdash \odot_n$ assures that n had started its life on the environment side. This bit of information is important as otherwise one could mistake the code $n\langle stop\rangle$ for the code of a (deadlocked) outgoing call. If $\Delta \vdash \odot_n$ and $n\langle stop\rangle$ is part of the component code, it is assured that the thread either has never actively entered the component before (and does so right now) or has left the component to the environment by some last outgoing return. In either case, the incoming call is possible now, and in both cases we can use $\odot_n$ as representative of the caller's identity.

The last call rule $\textsc{CallI}_3$ deals with the situation, that the thread n enters the component for the first time. This is assured by the premise $\Sigma' \vdash n : thread$. As in $\textsc{CallI}_2$, we do not have an indication from which clique the call originates, since the corresponding thread is *new*. What is assured is that the new thread has been created at some point before as instance of some environment thread class —otherwise the cross-border instantiation would have been observed and the thread name would not be fresh now— and by some environment clique. Indeed, *any* existing environment clique is a candidate that might have created the thread n. So the update to $\acute{\Xi}$ *non-deterministically guesses* to which environment clique the thread's origin $\odot_n$ belongs to. Note that $\odot_{\Sigma'}$ contains $\odot_n$ since $\Sigma' \vdash n$, which means $\acute{\Delta} \vdash \odot_n$ after the call.

For incoming thread creation in rule $\textsc{SpawnI}$, we need again to know the origin of the call, i.e., the spawning clique. The situation is similar to the one for $\textsc{CallI}_3$, in that the origin of the communication needs to be guessed. In the case of $\textsc{CallI}_3$, we use $\odot_n$ covering the situation where no actual calling object may be the source. Different from the situation of unknown caller is that here we obviously can not use $\odot_n$; that identity is incorporated into the *component* after the call. What is clear is that the spawner must be part of the environment prior to the call, i.e., $\Delta \vdash o_s$, where o_s might be some $\odot_{n'}$, i.e., a virtual clique of objects from which no actually existing objects have yet escaped to the component. Note that if $o_s = \odot_{n'}$, $\Delta \vdash o_s$ assures that $n \neq n'$. Note further that the name of the spawned thread is treated specifically in the definition of context update (cf. Definition 2 and 3) to cater for cross-border instantiation of the new thread. An incoming spawn action without known external objects is possible only in the very first step.

The remaining rules deal with outgoing communication and are simpler, as the "check-part" is omitted: With the code of the program present, the checks are guaranteed to be satisfied.

In addition to the external steps of Table 3, there are similar ones for communication via returns, and rules dealing with initial steps. They are included in the technical report [5].

$$\frac{\begin{array}{c} dom(\Phi') \subseteq fn(\lfloor a \rfloor) \quad \acute{\Xi} = \Xi + o_s \xrightarrow{a} o_r \quad \acute{\Xi} \vdash o_s \stackrel{\lfloor a \rfloor}{\to} o_r : T \quad \acute{\Theta} \vdash o_r \\ a = \nu(\Phi').\ n\langle call\ o_r.l(\vec{v})\rangle? \quad t_{blocked} = let\, x':T' = o\ blocks\ for\ o_s\ in\, t \end{array}}{\begin{array}{l} \Delta, \Sigma; E_\Delta \vdash \nu(\Phi).(C \parallel n\langle t_{blocked}\rangle) : \Theta, \Sigma; E_\Theta \xrightarrow{a} \\ \acute{\Delta}, \acute{\Sigma}; \acute{E}_\Delta \vdash \nu(\Phi).(C \parallel C(\Theta', \Sigma') \parallel n\langle let\, x:T = o_r.l(\vec{v})\ in\, o_r\ returns\ to\, o_s\ x; t_{blocked}\rangle) : \acute{\Theta}, \acute{\Sigma}; \acute{E}_\Theta \end{array}} \ \textsc{CallI}_1$$

$$\frac{\begin{array}{c} dom(\Phi') \subseteq fn(\lfloor a \rfloor) \quad \acute{\Xi} = \Xi + \odot_n \xrightarrow{a} o_r \quad \acute{\Xi} \vdash \odot_n \stackrel{\lfloor a \rfloor}{\to} o_r : T \quad \acute{\Theta} \vdash o_r \\ a = \nu(\Phi').\ n\langle call\ o_r.l(\vec{v})\rangle? \quad \Delta \vdash \odot_n \end{array}}{\begin{array}{l} \Delta, \Sigma; E_\Delta \vdash C \parallel n\langle stop\rangle : \Theta, \Sigma; E_\Theta \xrightarrow{a} \\ \acute{\Delta}, \acute{\Sigma}; \acute{E}_\Delta \vdash C \parallel C(\Theta', \Sigma') \parallel n\langle let\, x:T = o_r.l(\vec{v})\ in\, o_r\ returns\ to\ \odot_n\ x; stop\rangle : \acute{\Theta}, \acute{\Sigma}; \acute{E}_\Theta \end{array}} \ \textsc{CallI}_2$$

$$\frac{\begin{array}{c} dom(\Phi') \subseteq fn(\lfloor a \rfloor) \quad \acute{\Xi} = \Xi + o \xrightarrow{a} o_r \quad \acute{\Xi} \vdash \odot_n \stackrel{\lfloor a \rfloor}{\to} o_r : T \quad \acute{\Theta} \vdash o_r \\ a = \nu(\Phi').\ n\langle call\ o_r.l(\vec{v})\rangle? \quad \Delta \vdash o \quad \Sigma' \vdash n : thread \end{array}}{\begin{array}{l} \Delta, \Sigma; E_\Delta \vdash C : \Theta, \Sigma; E_\Theta \xrightarrow{a} \\ \acute{\Delta}, \acute{\Sigma}; \acute{E}_\Delta \vdash C \parallel C(\Theta', \Sigma' \setminus n) \parallel n\langle let\, x:T = o_r.l(\vec{v})\ in\, o_r\ returns\ to\ \odot_n\ x; stop\rangle : \acute{\Theta}, \acute{\Sigma}; \acute{E}_\Theta \end{array}} \ \textsc{CallI}_3$$

$$\frac{\begin{array}{c} dom(\Phi') \subseteq fn(\lfloor a \rfloor) \quad \acute{\Xi} = \Xi + o_s \xrightarrow{a} \odot_n \quad \acute{\Xi} \vdash o_s \stackrel{\lfloor a \rfloor}{\to} \odot_n : thread \\ a = \nu(\Phi').\langle spawn\, n\ of\ c_t(\vec{v})\rangle? \quad \acute{\Theta} \vdash o_r \quad \Delta \vdash o_s \quad \Theta \vdash c_t \quad \Sigma' \vdash n : thread \end{array}}{\Delta, \Sigma; E_\Delta \vdash C : \Theta, \Sigma; E_\Theta \xrightarrow{a} \acute{\Delta}, \acute{\Sigma}; \acute{E}_\Delta \vdash C \parallel C(\Theta', \Sigma' \setminus n) \parallel n\langle c_t(\vec{v})\rangle : \acute{\Theta}, \acute{\Sigma}; \acute{E}_\Theta} \ \textsc{SpawnI}$$

$$\frac{a = \nu(\Phi').\ n\langle call\ o_r.l(\vec{v})\rangle! \quad \Phi' = fn(\lfloor a \rfloor) \cap \Phi \quad \acute{\Phi} = \Phi \setminus \Phi' \quad \acute{\Delta} \vdash o_r \quad \acute{\Xi} = \Xi + o_s \xrightarrow{a} o_r}{\begin{array}{c} \Delta, \Sigma; E_\Delta \vdash \nu(\Phi).(C \parallel n\langle let\, x:T = o_s\ o_r.l(\vec{v})\ in\, t\rangle) : \Theta, \Sigma; E_\Theta \xrightarrow{a} \\ \acute{\Delta}, \acute{\Sigma}; \acute{E}_\Delta \vdash \nu(\acute{\Phi}).(C \parallel n\langle let\, x:T = o_s\ blocks\ for\ o_r\ in\, t\rangle) : \acute{\Theta}, \acute{\Sigma}; \acute{E}_\Theta \end{array}} \ \textsc{CallO}$$

$$\frac{\begin{array}{c} a = \nu(\Phi').\langle spawn\, n'\ of\ c_t(\vec{v})\rangle! \quad \Phi' = (fn(\lfloor a \rfloor) \setminus n') \cap \Phi \quad \acute{\Phi} = \Phi \setminus \Phi' \\ \Delta \vdash c_t \quad \acute{\Xi} = \Xi + o_s \xrightarrow{a} \odot_{n'} \end{array}}{\begin{array}{l} \Delta, \Sigma; E_\Delta \vdash \nu(\Phi).(C \parallel n\langle let\, x:T = o_s\ spawn\ c_t(\vec{v})\ in\, t\rangle) : \Theta, \Sigma; E_\Theta \xrightarrow{a} \\ \acute{\Delta}, \acute{\Sigma}; \acute{E}_\Delta \vdash \nu(\acute{\Phi}).(C \parallel n\langle let\, x:T = n'\ in\, t\rangle) : \acute{\Theta}, \acute{\Sigma}; \acute{E}_\Theta \end{array}} \ \textsc{SpawnO}$$

Table 3. External steps

3 Legal traces

In this section we present a proof system which provides an independent characterization of which traces are possible as interface behavior between component and environment. We call those traces legal. "Half" of the work has been done already by the careful design of the open semantics of Section 2.2.4, where the absent environment is represented abstractly by the name and connectivity contexts. For characterizing the legal traces, we analogously abstract away from the program code, which makes the system completely symmetric. Remember that the assumption and commitment contexts in the operational semantics where used asymmetrically insofar, as the commitment contexts where updated as overapproximation of the actual component, but not used in *checking* whether

$$\dfrac{\begin{array}{c} a = \nu(\Phi').\ n\langle call\ o_r.l(\vec{v})\rangle? \quad \vdash r \rhd o_s \xrightarrow{a} o_r \\ \acute{\Xi} = \bigoplus \Xi_i + a \quad \Theta_i \vdash o_r \quad \epsilon \neq a_i = (a, o_r) \downarrow_{\Theta_i} \quad \acute{\Delta}, \acute{\Sigma} \vdash r\ a \rhd s : trace\ \acute{\Theta}, \acute{\Sigma} \end{array}}{\Delta_1, \Sigma_1 \vdash r \rhd a_1\ s : trace\ \Theta_1, \Sigma_1 \quad \ldots \quad \Delta_k, \Sigma_k \vdash r \rhd a_k\ s : trace\ \Theta_k, \Sigma_k} \text{ L-CALLI}$$

$$\dfrac{\Delta, \Sigma \vdash ra \rhd s : trace\ \Theta, \Sigma \quad a = \gamma? \quad \vdash r \rhd o_s \xrightarrow{a} o_r \quad \Theta \not\vdash o_r}{\Delta, \Sigma \vdash r \rhd s : trace\ \Theta, \Sigma} \text{ L-SKIPI}$$

Table 4. Legal traces, branching on Θ

a component step, i.e., an outgoing communication, is possible as next interaction.

3.1 A branching derivation system characterizing legal traces

Unlike the treatment in the operational semantics, the connectivity of objects is not explicitly represented by connectivity contexts; instead, the *tree structure* of the derivation itself represents the connectivity and its change. There are two variants of the derivation system, one from the perspective of the *component,* and one from the perspective of the *environment.* Each derivation corresponds to a *forest,* with each tree representing a component, respectively environment clique at the end. The judgments are of the form

$$\Delta, \Sigma \vdash_{\Theta} r \rhd s : trace\ \Theta, \Sigma \tag{5}$$

where r represents the history or past interaction, and s the future interaction. We write $\vdash_{\Theta}$ to indicate that legality is checked from the perspective of the component. From that perspective, we maintain as invariant that on the commitment side, the context Θ represents one single clique. Thus the connectivity among objects of Θ needs no longer be remembered. What needs to be remembered still are the thread names known by Θ and the cross-border object connectivity, i.e., the acquaintance of the clique represented by Θ with objects of the environment. This information is kept in Δ resp. Σ. Note that this corresponds to the environmental objects mentioned in $E_{\Theta} \subseteq \Theta \times (\Theta + \Delta + \Sigma)$, projected onto the component clique under consideration, in the linear system.

The connectivity of the environment is *ignored* which implies that the system of Table 4 *cannot* assure that the environment behaves according to a possible connectivity. On the other hand, dualizing the rules checks whether the environment adheres to possible connectivity.

Now to the rules of Table 4. As before, rule L-CALLI deals with incoming calls. The call is possible only when the thread is input call enabled after the current history. This is checked by the premise $\vdash r \rhd o_s \xrightarrow{a} o_r : ok$, which also determines caller and callee. We omit the definition of $\vdash r \rhd o_s \xrightarrow{a} o_r : ok$, characterizing enabledness of a after trace r. The definition is the straightforward extension of the one from [4] to a multi-threaded setting.

Since from the perspective of the component, the connectivity of the environment is no longer represented as assumption, there are *no* premises checking connectivity! An interesting part concerns the treatment of the commitment context: Incoming communication may *update* the component connectivity, in that new cliques may be created or existing cliques may merge. The merging of component cliques is now represented by a branching of the proof system. Leaves of the resulting tree (respectively forest) correspond to freshly created cliques.

In rule L-CallI, the context Θ in the premise corresponds to the merged clique, the Θ_i below the line to the still split cliques before the merge. The Θ_i's form a partitioning of the component objects before the communication, Θ is the disjoint combination of the Θ_i's plus the lazily instantiated objects from Θ'. For the cross-border connectivity, i.e., the environmental objects known by the component cliques, the different component cliques Θ_i may of course share acquaintance; thus, the parts Δ_i and Σ_i are not merged disjointly, but by ordinary "set" union.[5] These restrictions are covered by the definition of the (partial) operation $\bigoplus \Xi_i$.

We omit the rules dealing with incoming returns and incoming spawn labels, and furthermore those for outgoing communication.

The skip-rules stipulate that an action a which does not belong to the component clique under consideration, is omitted from the component's "future" (interpreting the rule from bottom to top). The distinction is made according to the sender resp. the receiver of the communication (cf. rule L-SkipO resp. L-SkipI).

Definition 4 (Legal traces, branching system). *We write $\Delta \vdash_\Theta t : trace\,\Theta$, if there exists a derivation forest using the rules of Table 4 with roots $\Delta_i, \Sigma_i \vdash t \rhd \epsilon : trace\,\Theta_i, \Sigma_i$ and a leaf justified by one of the initial rules* L-CallI$_0$ *or* L-CallO$_0$. *Using the dual rules, we write $\vdash_\Delta$ instead of $\vdash_\Theta$.*

We write $\Delta \vdash_{\Delta\wedge\Theta} t : trace\,\Theta$, if there exits a pair of derivations in the $\vdash_\Delta$- and the $\vdash_\Theta$- system with a consistent pair of root judgments.

[5] Technically, of course, the contexts are syntactical entities of the calculus and not sets; however, the invariants enforced by the type system and maintained by the semantics allows to consider them as finite mappings from names to types.

$$\frac{a = \nu(\Phi').\ n\langle call\ o_r.l(\vec{v})\rangle? \quad \vdash r \rhd o_s \xrightarrow{a} o_r \quad \acute{\Delta}, \acute{\Sigma}, \acute{\Theta} \vdash \lfloor a \rfloor : ok \qquad \Delta \not\vdash static \quad \acute{\Delta} \vdash o_s \quad \acute{\Xi} = \Xi + a \quad \acute{\Delta}, \acute{\Sigma} \vdash r\ a \rhd s : trace\,\acute{\Theta}, \acute{\Sigma}}{\Delta, \Sigma \vdash r \rhd a\ s : trace\,\Theta, \Sigma}\ \text{L-CallI}$$

$$\frac{\acute{\Delta} = \Delta \not\vdash o_s \quad \Delta, \Sigma \vdash ra \rhd s : trace\,\Theta, \Sigma \quad a = \gamma? \quad \vdash r \rhd o_s \xrightarrow{a} o_r}{\Delta, \Sigma \vdash r \rhd s : trace\,\Theta, \Sigma}\ \text{L-SkipI}$$

Table 5. Legal traces, branching on Δ

To accommodate for the simpler structure of the contexts, we adopt the notational conventions (cf. Notation 1) appropriately.

The way a communication step updates the name context can be defined as simplification of the treatment in the operational semantics (cf. Definition 2). As before we write $\Phi + a$ for the update.

3.2 Soundness of the abstractions

The section contains the basic soundness results of the abstractions,

With E_Δ and E_Θ as part of the judgment, we must still clarify what it "means", i.e., when does $\Delta, \Sigma; E_\Delta \vdash C : \Theta, \Sigma; E_\Theta$ hold? The relation E_Θ asserts about the component C that the connectivity of the objects from the component is *not larger than* the connectivity entailed by E_Θ. Given a component C and two names o from Θ and n from $\Theta + \Delta + \Sigma$, we write $C \vdash o \hookrightarrow n$, if $C \equiv \nu(\Phi).(C' \parallel o[\ldots, f = n, \ldots])$ where o and n are not bound by Φ, i.e., o contains in one of its fields a reference to n. We can thus define:

Definition 5. *The judgment $\Delta, \Sigma; E_\Delta \vdash C : \Theta, \Sigma; E_\Theta$ holds, if $\Delta, \Sigma \vdash C : \Theta, \Sigma$, and if $C \vdash n_1 \hookrightarrow n_2$, then $\Theta, \Sigma; E_\Theta \vdash n_1 \leftrightharpoons\!\!\hookrightarrow n_2 : \Delta, \Sigma$.*

We often simply write $\Delta, \Sigma; E_\Delta \vdash C : \Theta, \Sigma; E_\Theta$ to assert that the judgment is satisfied. Note that references mentioned in threads do not "count" as acquaintance.

Lemma 1 (Subject reduction). $\Delta, \Sigma; E_\Delta \vdash C : \Theta, \Sigma; E_\Theta \stackrel{s}{\Longrightarrow} \acute{\Delta}, \acute{\Sigma}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}, \acute{\Sigma}; \acute{E}_\Theta$*, then* $\acute{\Delta}, \acute{\Sigma} \vdash \acute{C} : \acute{\Theta}, \acute{\Sigma}$*. A fortiori: If* $\Delta, \Sigma, \Theta \vdash n : T$*, then* $\acute{\Delta}, \acute{\Sigma}, \acute{\Theta} \vdash n : T$*.*

Besides the static abstraction of the type system, also the assertions about the heap topology (cf. Definition 5) preserved.

Lemma 2 (Soundness of the connectivity abstraction). $\Delta, \Sigma; E_\Delta \vdash C : \Theta, \Sigma; E_\Theta \stackrel{s}{\Longrightarrow} \acute{\Delta}, \acute{\Sigma}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}, \acute{\Sigma}; \acute{E}_\Theta$*, then* $\acute{\Delta}, \acute{\Sigma}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}, \acute{\Sigma}; \acute{E}_\Theta$*.*

An interesting invariant concerns the connectivity of names transmitted boundedly. Incoming communication, e.g., not only updates the commitment contexts —something one would expect— but also the *assumption* contexts. The fact that no new information is learnt about already known objects ("no surprise") in the assumptions can be phrased using the notion of conservative extension.

Definition 6 (Conservative extension). *Given two pairs (Φ, E_Δ) and $(\acute{\Phi}, \acute{E}_\Delta)$ of name context and connectivity context, i.e., $E_\Delta \subseteq \Phi \times \Phi$ (and analogously for $(\acute{\Phi}, \acute{E}_\Delta)$), we write $(\Phi, E_\Delta) \vdash (\acute{\Phi}, \acute{E}_\Delta)$ if the following two conditions holds:*

1. *$\acute{\Phi} \vdash \Phi$ and*
2. *$\acute{\Phi} \vdash n_1 \leftrightharpoons n_2$ implies $\Phi \vdash n_1 \leftrightharpoons n_2$, for all n_1, n_2 with $\Phi \vdash n_1, n_2$.*

Lemma 3 (No surprise). *Let $\Delta, \Sigma; E_\Delta \vdash C : \Theta, \Sigma; E_\Theta \xrightarrow{a} \acute{\Delta}, \acute{\Sigma}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}, \acute{\Sigma}; \acute{E}_\Theta$ for some incoming label a. Then $\Delta, \Sigma; E_\Delta \vdash \acute{\Delta}, \acute{\Sigma}; \acute{E}_\Delta$. For outgoing steps, the situation is dual.*

Lemma 4 (Soundness of legal trace system). *If $\Delta_0; \vdash C : \Theta_0;$ and $\Delta_0; \vdash C : \Theta_0; \stackrel{t}{\Longrightarrow}$, then $\Delta_0 \vdash t : trace\, \Theta_0$.*

4 Conclusion

Related work [13] presents a fully abstract model for *Object-Z*, an object-oriented extension of the *Z* specification language. It is based on a refinement of the simple trace semantics called the complete-readiness model, which is related to the readiness model of Olderog and Hoare. [14] investigates full abstraction in an object calculus with subtyping. The setting is slightly different from the one here, as the paper does not compare a contextual semantics with a denotational one, but a semantics by translation with a direct one. The paper considers neither concurrency nor aliasing. Recently, Jeffrey and Rathke [12] extended their work [11] on trace-based semantics from an object-based setting to a core of *Java*, called *JavaJr*, including classes and subtyping. However, their semantics avoids the issue of object connectivity by using a notion of *package*. [7] tackles the problem of full abstraction and observable component behavior and connectivity in a UML-setting.

Future work We plan to extend the language with further features to make it more resembling *Java* or $C^{\#}$. Concerning the concurrency model, objects should be extended by lock-*synchronization* as provided by *Java*'s `synchronized` methods, and furthermore monitor synchronization via wait- and signal-methods. Another interesting direction for extension concerns the type system, in particular to include *subtyping* and *inheritance*. This is challenging especially if the component may inherit from environment classes and vice versa. For a first step in this direction we will concentrate on subtyping alone, i.e., relax the type discipline of the calculus to subtype polymorphism, but without inheritance. Another direction is to extend the semantics to a *compositional* one; currently, the semantics is open in that it is defined in the context of an environment. However, general composition of open program fragments is not defined. Finally, we work on adapting the full abstraction proof of [3] to the new setting, i.e., to deal with thread classes. The results of Section 3.2 are covering the soundness-part of the full-abstraction result.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects.* Monographs in Computer Science. Springer, 1996.
2. E. Ábrahám, M. M. Bonsangue, F. S. de Boer, and M. Steffen. A structural operational semantics for a concurrent class calculus. Technical Report 0307, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel, Aug. 2003.
3. E. Ábrahám, M. M. Bonsangue, F. S. de Boer, and M. Steffen. Object connectivity and full abstraction for a concurrent calculus of classes. In Z. Li, editor, *ICTAC'04*, volume 3407 of *Lecture Notes in Computer Science*, pages 38–52. Springer-Verlag, July 2004.
4. E. Ábrahám, F. S. de Boer, M. M. Bonsangue, A. Grüner, and M. Steffen. Observability, connectivity, and replay in a sequential calculus of classes. In Bosangue et al. [6]. To appear.

5. E. Ábrahám, A. Grüner, and M. Steffen. An open structural operational semantics for an object-oriented calculus with thread classes. Technical Report 0505, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel, May 2005.
6. M. Bosangue, F. S. de Boer, W.-P. de Roever, and S. Graf, editors. *Proceedings of the Third International Symposium on Formal Methods for Components and Objects (FMCO 2004)*, Lecture Notes in Computer Science. Springer-Verlag, 2005. To appear.
7. F. S. de Boer, M. Bonsangue, M. Steffen, and E. Ábrahám. A fully abstract trace semantics for UML components. In Bosangue et al. [6]. To appear.
8. ECMA International Standardizing Information and Communication Systems. *C*$^{\#}$ *Language Specification*, 2nd edition, Dec. 2002. Standard ECMA-334.
9. A. D. Gordon and P. D. Hankin. A concurrent object calculus: Reduction and typing. In U. Nestmann and B. C. Pierce, editors, *Proceedings of HLCL '98*, volume 16.3 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 1998.
10. J. Gosling, B. Joy, G. L. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, Second edition, 2000.
11. A. Jeffrey and J. Rathke. A fully abstract may testing semantics for concurrent objects. In *Proceedings of LICS '02*. IEEE, Computer Society Press, July 2002.
12. A. Jeffrey and J. Rathke. Java Jr.: A fully abstract trace semantics for a core Java language. In M. Sagiv, editor, *Proceedings of ESOP 2005*, volume 3444 of *Lecture Notes in Computer Science*, pages 423–438. Springer-Verlag, 2005.
13. G. P. Smith. *An Object-Oriented Approach to Formal Specification*. PhD thesis, Department of Computer Science, University of Queensland, Oct. 1992.
14. R. Viswanathan. Full abstraction for first-order objects with recursive types and subtyping. In *Proceedings of LICS '98*. IEEE, Computer Society Press, July 1998.