Abstract Local Reasoning for Concurrent Libraries: Mind the Gap

Philippa Gardner, Azalea Raad, Mark Wheelhouse, Adam Wright[1]

Department of Computing, Imperial College London, UK

Abstract

We study abstract local reasoning for concurrent libraries. There are two main approaches: provide a specification of a library by *abstracting* from concrete reasoning about an implementation; or provide a direct abstract library specification, justified by *refining* to an implementation. Both approaches have a significant gap in their reasoning, due to a mismatch between the *abstract connectivity* of the abstract data structures and the *concrete connectivity* of the concrete heap representations. We demonstrate this gap using structural separation logic (SSL) for specifying a concurrent tree library and concurrent abstract predicates (CAP) for reasoning about a concrete tree implementation. The gap between the abstract and concrete connectivity emerges as a mismatch between the SSL tree predicates and CAP heap predicates. This gap is closed by an *interface function* I which links the abstract and concrete connectivity. In the accompanying technical report, we generalise our SSL reasoning and results to arbitrary concurrent data libraries.

Keywords: Concurrency, Abstraction, Refinement, Separation, Translation, Reasoning

1 Introduction

Local reasoning was first introduced in separation logic to reason about the RAM memory model. Since then, there has been considerable work on combining local reasoning with abstraction. There are two main approaches. One approach starts with concrete reasoning about code that manipulates the standard heap and then builds up layers of *abstraction*: this approach is used by concurrent abstract predicates (CAP) and its variants [1,16,17], and is ideal for reasoning about the concurrent library `java.util.concurrent` where the library functions are built up using implementations. The other approach provides direct abstract specifications of abstract code manipulating abstract models, and then justifies the specification by *refining* it to a correct concrete implementation: this approach is used by context logic [7,9], and is ideal for reasoning about libraries such as POSIX and sequential DOM where the library specification is not grounded on implementation.

These current abstraction and refinement approaches have a significant gap in their reasoning. With abstraction, the implementation details leak into the ab-

[1] Email: `{pg, azalea, mjw03, adw07}@doc.ic.ac.uk`

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

straction. For example, consider a CAP predicate $\mathsf{tree}(t)(i,j)(l,u,r)$ for describing tree fragments. The predicate is parameterised by an abstract tree t, with concrete pointers $(i,j)(l,u,r)$ describing the concrete interface used to connect the concrete tree fragments. In this case, the concrete interface consists of the first (i) and last (j) nodes of the tree fragment, and the parent (u), left (l) and right (r) siblings. A different implementation, say one using lists, would require a different concrete interface. Thus, this tree predicate is not abstract enough to reason abstractly about updating tree fragments [2]. The missing piece is an abstract way of splitting and combining tree fragments, and a way of linking it to the concrete interface given by $(i,j)(l,u,r)$. With the refinement approach, we have examples where the specification is truly abstract (e.g. [12,19]), but they are typically justified by a soundness result to an operational semantics rather than an implementation. For example, truly abstract reasoning of a tree module (such as DOM) works with predicates based on connecting tree fragments using contexts and place holders. In contrast, the concrete reasoning about an implementation uses pointers. We need to bridge the gap between the *abstract connectivity* of abstract data structures and the *concrete connectivity* of concrete heap representations.

We introduce *Structural Separation Logic (SSL)* for reasoning about concurrent abstract data libraries [2,19]. SSL is underpinned by a particular general approach to abstract connectivity for structured data. In this paper, we use a simple concurrent tree library to illustrate our ideas. We provide concrete reasoning about a tree implementation of the library using CAP [1]. The gap between the abstract and concrete connectivity emerges as a mismatch between SSL tree predicates and CAP tree predicates. This gap is closed by an *interface function I* which links the abstract and concrete connectivity. In the accompanying technical report [18] and Raad's forthcoming thesis, we generalise SSL and our results to arbitrary structured data libraries. The work presented here does depend on the particular SSL approach to abstract connectivity. Our ideas should, however, apply whenever there is a mismatch between abstract and concrete connectivity.

SSL supports reasoning about fine-grained abstract data fragments stored in *abstract heaps*. Abstract heaps contain cells identified by abstract addresses (e.g. address $\mathbf{x}$) whose values are the disjoint data fragments. These data fragments contain context holes, also given by abstract addresses, which are place holders for the data fragments found at the appropriate abstract cells. For example, the SSL predicate $\mathsf{ATree}\,(t)(\mathbf{x})$ describes a tree cell with abstract address $\mathbf{x}$ containing tree context t. We can split (abstractly allocate) this predicate to obtain the semantically-equivalent assertion $\exists \mathbf{y}.\, \mathsf{ATree}\,(t_1)(\mathbf{x}) * \mathsf{ATree}\,(t_2)(\mathbf{y})$ with $t = t_1[t_2/\mathbf{y}]$. The assertion represents the same underlying tree fragment, just in two disjoint parts. Reasoning about semantically-equivalent assertions is only possible due to recent advances in concurrent reasoning given by the Views framework [6].

We provide a natural implementation of our tree library and show that it is correct with respect to our abstract SSL specification, by relating abstract SSL

[2] The same issue arises with the well-known predicate $\mathsf{listseg}([1,2,3])(i)(r)$ which describes list fragments with concrete address i, abstract contents $1,2,3$, and concrete right pointer r. This predicate is appropriate for implementations using singly-linked lists, but not for those using doubly-linked lists which require an additional concrete left pointer. The predicate is not abstract enough because the concrete interface is leaking into the predicate. The missing bit is the abstract connectivity of the list fragment.

specifications of the library functions with CAP-like specifications of the concrete function implementations. To do this, we must extend CAP predicates with a *interface function* I which relates abstract addresses with concrete pointers. For example, consider the abstract tree predicate $\mathsf{ATree}(t)(\mathbf{x})$ and the corresponding concrete CAP predicate $\mathsf{CTree}^{I}(t)(x)$, where the interface function I relates the abstract address $\mathbf{x}$ with the pointer interface $(i,j)(l,u,r)$. We prove that our implementation is correct with respect to our abstract specification, using a $*$-preserving translation (analogous to *locality-preserving* translation in [9]) *parameterised* by I.

In this paper, we concentrate on refinement. It is trivial to adapt our work to the approach of fictional separation logic (FSL) [13], where translations are incorporated within the Hoare derivation. However, our choice of translation differs fundamentally from FSL. FSL is designed to reason about *sequential* programs using $*$-*breaking* translations (analogous to *locality-breaking* translation in [9]): the proofs of FSL assume *all* possible frames are preserved during the execution of program. While this is a reasonable assumption when reasoning about sequential programs, it is non-trivial to establish its soundness for concurrent programs. One possible way to demonstrate its correctness is to provide linearisability proofs to show that all frames are indeed preserved and that the program behaves atomically. In contrast, our $*$-preserving translation ensures that compatible stable resources at the abstract level $(p * q)$ yield compatible stable resources once translated $(p' * q')$ [3]. The correctness of concurrent programs then follows immediately from the disjoint concurrency rule of concurrent separation logic [20].

With abstraction using CAP, we start by concretely reasoning about the heap and then build up levels of abstraction in such a way that $*$ is preserved. As demonstrated, current CAP reasoning leaks implementation details into the abstractions. We believe this can be rectified by extending the abstraction rule to hide the interface functions as well as the predicate interpretations.

We believe that interface functions I have been barely studied in the literature. They do appear in [9] for reasoning about *sequential* libraries using context logic. Context logic is not fine-grained enough to extend to concurrency, since it uses a non-commutative separating application unsuitable for use with the disjoint concurrency rule. SSL reasoning makes this extension possible. SSL is used in [19] for specifying sequential POSIX, with the aim to extend to concurrent POSIX in future. However, soundness was proved by comparison with an abstract operational semantics, so the relationship between abstract and concrete connectivity did not arise. In [15,16,13], there is an emphasis on interpretations which relate abstract and concrete states. However, there is no mention of a mapping I between abstract and concrete interfaces since the connectivity in the examples studied is simple.

2 Structural Separation Logic: Tree Library

Structural separation logic (SSL) is a general program logic for specifying structured data libraries and reasoning locally about client programs which call such libraries. Here, we give the intuition and technical details of SSL using an abstract tree library.

[3] Note that $p * q$ does not necessarily suggest physically *disjoint* resources; rather two *compatible* resources that can be composed together providing a *fiction of disjointness*.

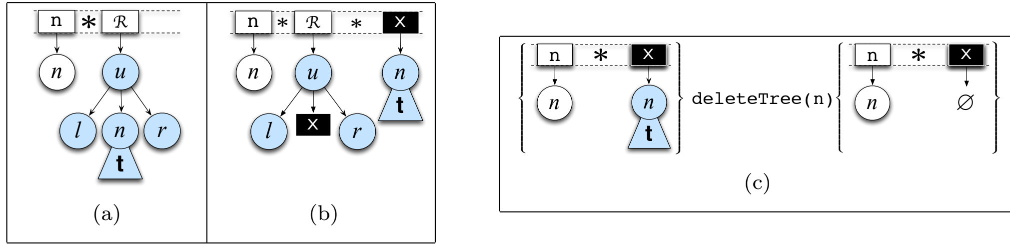

Fig. 1. Abstract (de)allocation in SSL (left); reasoning about `deleteTree(n)` in SSL (right)

We give the general theory of SSL in the accompanying technical report [18]. Further details, including a wide number of examples, can be found in [2].

2.1 Intuition

We give our axiomatic SSL specification of a simple `deleteTree(n)` command. Intuitively, this command removes the entire subtree whose top node identifier corresponds to the value of variable `n`, leaving the rest of the tree unchanged. We formalise this English description using assertions which describe *abstract heaps*.

Abstract heaps store abstract data fragments. For instance, Fig. 1a illustrates an abstract heap describing a variable cell `n` with value n, and a tree cell $\mathcal{R}$ with a complete abstract tree as its value. This tree consists of a subtree $n[t]$ with parent u, and left and right siblings l and r with no children. It abstracts away from how a tree might be concretely represented in a machine.

Intuitively, the `deleteTree(n)` command only affects the subtree identified by n. Abstract heaps enable structured data to be *split* to provide direct access to this subtree by imposing additional instrumentation using *abstract addresses*. Consider the transition from Fig. 1a to 1b. Fig. 1a contains an abstract heap with a complete tree at $\mathcal{R}$. We split this complete tree using *abstract allocation* to obtain the abstract heap in Fig. 1b with subtree $n[t]$ at a fresh, fictional *abstract cell* $\mathbf{x}$ and an incomplete tree at $\mathcal{R}$ with a *context hole* $\mathbf{x}$ indicating the position to which the subtree will return. The subtree at n can now be accessed directly. Once the updates have been achieved, the heap can be joined back together using *abstract deallocation*, as in the transition from Fig. 1b to 1a.

The axiomatic specification of `deleteTree(n)` (Fig. 1c) is formalised as:

$$\{\mathsf{var}\,(\mathtt{n}, n) \times \mathsf{ATree}\,(n[\mathit{isComplete}])(\mathbf{x})\}\ \mathtt{deleteTree(n)}\ \{\mathsf{var}\,(\mathtt{n}, n) \times \mathsf{ATree}\,(\varnothing)(\mathbf{x})\}$$

The precondition describes a variable store, in which variable `n` has value n and abstract cell $\mathbf{x}$ has *complete* subtree with top node n as its value [4]. Since the subtree at $\mathbf{x}$ is complete, we own the exclusive right to update its contents. Similarly, the postcondition states that the result of the update is an empty tree at abstract cell $\mathbf{x}$, while variable `n` remains unchanged. Note that the abstract cell $\mathbf{x}$ must be preserved in order to join this tree fragment with the tree it was split from, using abstract

[4] Note that the precondition is a pair consisting of a variable assertion and a subtree (heap) assertion. We use the variables-as-resource model [14]. However, in contrast to the assertions of variables-as-resource where heap and variable assertions are combined using $*$, we keep the two components separate. As we demonstrate in §4, this is because when translating a library, only heaps are transformed by the translation while program variables are simply preserved.

$\{(\mathsf{var}\,(\mathtt{l}, l) * \mathsf{var}\,(\mathtt{n}, n)) \times (\mathsf{ATree}\,(u[l \otimes n[t] \otimes r])(\mathcal{R}))\}$

//Abstract allocation (Twice)

$\{(\mathsf{var}\,(\mathtt{l}, l) * \mathsf{var}\,(\mathtt{n}, n)) \times (\exists \mathbf{x}, \mathbf{y}. \mathsf{ATree}\,(u[\mathbf{y} \otimes \mathbf{x} \otimes r])(\mathcal{R}) * \mathsf{ATree}\,(l)(\mathbf{y}) * \mathsf{ATree}\,(n[t])(\mathbf{x}))\}$

//Existential elimination and frame rule

$\{(\mathsf{var}\,(\mathtt{l}, l) * \mathsf{var}\,(\mathtt{n}, n)) \times (\mathsf{ATree}\,(l)(\mathbf{y}) * \mathsf{ATree}\,(n[t])(\mathbf{x}))\}$

//Disjoint Concurrency rule

$\{(\mathsf{var}\,(\mathtt{l}, l)) \times (\mathsf{ATree}\,(l)(\mathbf{y}))\}$	$\{(\mathsf{var}\,(\mathtt{n}, n)) \times (\mathsf{ATree}\,(n[t])(\mathbf{x}))\}$
`deleteTree(l)`	`deleteTree(n)`
$\{(\mathsf{var}\,(\mathtt{l}, l)) \times (\mathsf{ATree}\,(\varnothing)(\mathbf{y}))\}$	$\{(\mathsf{var}\,(\mathtt{n}, n)) \times (\mathsf{ATree}\,(\varnothing)(\mathbf{x}))\}$

$\{(\mathsf{var}\,(\mathtt{l}, l) * \mathsf{var}\,(\mathtt{n}, n)) \times (\mathsf{ATree}\,(\varnothing)(\mathbf{y}) * \mathsf{ATree}\,(\varnothing)(\mathbf{x}))\}$

//Existential elimination and frame rule

$\{(\mathsf{var}\,(\mathtt{l}, l) * \mathsf{var}\,(\mathtt{n}, n)) \times (\exists \mathbf{x}, \mathbf{y}. \mathsf{ATree}\,(u[\mathbf{y} \otimes \mathbf{x} \otimes r])(\mathcal{R}) * \mathsf{ATree}\,(\varnothing)(\mathbf{y}) * \mathsf{ATree}\,(\varnothing)(\mathbf{x}))\}$

//Abstract Deallocation (Twice)

$\{(\mathsf{var}\,(\mathtt{l}, l) * \mathsf{var}\,(\mathtt{n}, n)) \times (\mathsf{ATree}\,(u[r])(\mathcal{R}))\}$

Fig. 2. Proof derivation of the concurrent program `deleteTree(l) || deleteTree(n)`.

deallocation. The footprint of this command is *small* in the sense that it intuitively captures the minimum resources required for safe execution of `deleteTree(n)`.

With this axiomatisation, we can verify that the simple client program `deleteTree(l)` $\|$ `deleteTree(n)` deletes two disjoint subtrees concurrently. Consider the proof derivation in Fig. 2 with program variables `l` and `n`, their values l and n, and an abstract tree predicate $\mathsf{ATree}\,(u[l \otimes n[t] \otimes r])(\mathcal{R})$ denoting a complete tree at $\mathcal{R}$ containing a subtree with top node n, left and right siblings with top nodes l and r, and parent u. The initial precondition describes the complete tree at $\mathcal{R}$. To get at the two subtrees, we split the tree twice using abstract allocation, placing the subtrees at nodes l and n at the freshly allocated cell addresses $\mathbf{y}$ and $\mathbf{x}$, respectively. We apply the standard separation logic rules of existential elimination and frame to temporarily set aside the partial tree at $\mathcal{R}$ as it is not being updated by the program. The resulting state can then be split into two disjoint parts using the disjoint concurrency rule. Both parts are now in the right form to match the precondition of the `deleteTree` axiom. The two updates can happen resulting in the postconditions with empty trees at $\mathbf{y}$ and $\mathbf{x}$ which are joined together by the disjoint concurrency rule. We reintroduce the state set aside through the existential elimination and frame rule, and obtain the complete tree at $\mathcal{R}$ using abstract deallocation twice to remove $\mathbf{y}$ and $\mathbf{x}$.

2.2 Technical Details

We give the technical details of SSL using the tree library $\mathbb{T}$ discussed in §2.1.

Definition 2.1 The set of *abstract atomic tree commands* of $\mathbb{T}$ are defined as:

$$\textsc{Atom}_{\mathbb{T}} ::= \mathtt{m} := \mathtt{getFirst(n)} \mid \mathtt{m} := \mathtt{getRight(n)} \mid \mathtt{m} := \mathtt{getUp(n)}$$
$$\mid \mathtt{m} := \mathtt{newNodeAfter(n)} \mid \mathtt{deleteTree(n)} \mid \mathtt{appendChild(m,\ n)}$$

for all program variables $\mathtt{n}, \mathtt{m} \in \textsc{PVars}$. Our commands are chosen to demonstrate a wide range of structural manipulations. The command `getFirst(n)` returns the

identifier of the first child of node `n`, or `null` if `n` has no children. The `getRight(n)` and `getUp(n)` commands behave analogously with respect to the right sibling and parent of node `n`, respectively. The command `newNodeAfter(n)` creates a new node with a fresh identifier, making it the right sibling of node `n` and returning the fresh identifier. The `deleteTree(n)` command removes the entire subtree identified by `n` from the tree. Finally, the command `appendChild(m,n)` moves the subtree at `n` to be the last child of node `m`. Each of these commands *faults* if any of the nodes given as parameters are not present in the tree. Additionally, the `appendChild(m,n)` command faults if `m` is a descendant of `n`. These commands are intended to be used with any programming language. In this paper, we use the programming language of the Views framework [6] instantiated with $\textsc{Atom}_{\mathbb{T}}$ as the set of atomic commands. We write $\textsc{Prog}_{\mathbb{T}}$ to denote the set of programs written in this language.

We specify the tree commands using abstract heaps with cells whose addresses are either abstract addresses or the tree root address $\mathcal{R}$, and whose values are abstract trees.

Definition 2.2 Given a countably infinite set of abstract addresses $\textsc{AAdd}$ ranged over by $\mathbf{x}, \mathbf{y}, \mathbf{z}$, the set of addresses for the abstract tree heaps is $\textsc{Add}_{\mathbb{T}} \triangleq \{\mathcal{R}\} \uplus \textsc{AAdd}$.

We now define abstract trees, which can either be complete trees or tree fragments with abstract addresses for context holes [5].

Definition 2.3 Given the set of abstract addresses $\textsc{AAdd}$, the set of *abstract trees* $\textsc{Data}_{\mathbb{T}}$, ranged over by $t, t_1, \cdots t_n$, is defined inductively as:

$$t ::= \varnothing \mid n[t] \mid t_1 \otimes t_2 \mid \mathbf{x}$$

where $n \in \mathbb{N}^+$, $\mathbf{x} \in \textsc{AAdd}$, the sets $\textsc{AAdd}$ and $\mathbb{N}^+$ are disjoint, and no tree contains duplicate node identifiers or abstract addresses. Abstract trees are equal up to the associativity of $\otimes$ with unit $\varnothing$. For brevity, we write n for $n[\varnothing]$. There is an associated *identifiers* function $ids : \textsc{Data}_{\mathbb{T}} \to \wp(\mathbb{N}^+)$ and an associated *addresses* function $addrs : \textsc{Data}_{\mathbb{T}} \to \wp(\textsc{AAdd})$ which respectively, return the sets of node identifiers and abstract addresses present in an abstract tree. The application $t_1 \circ_{\mathbf{x}} t_2$ is standard: it is undefined when $\mathbf{x} \notin addrs(t_1)$ and is otherwise defined as $t_1[t_2/\mathbf{x}]$, denoting the standard substitution of t_2 for $\mathbf{x}$ in t_1.

An abstract tree heap is a mapping from addresses to abstract trees.

Definition 2.4 The set of *abstract tree heaps* $H_{\mathbb{T}} : \textsc{Add}_{\mathbb{T}} \overset{\text{fin}}{\rightharpoonup} \textsc{Data}_{\mathbb{T}}$, ranged over by $h, h_1, \cdots, h_n$, is the set of functions from addresses to abstract trees such that for all $h \in H_{\mathbb{T}}$ the following restrictions hold:

$$\forall a_1, a_2 \in \textsc{Add}_{\mathbb{T}}.\ a_1 = a_2 \vee addrs(h(a_1)) \cap addrs(h(a_2)) = \varnothing$$
$$\forall a_1, a_2 \in \textsc{Add}_{\mathbb{T}}.\ a_1 = a_2 \ \vee\ ids(h(a_1)) \cap ids(h(a_2)) = \varnothing$$
$$\nexists\, \mathbf{x} \in \textsc{AAdd}.\ \mathbf{x}\ D_h^+\ \mathbf{x} \ \ \wedge\ \ \forall \mathbf{x} \in \text{dom}(h) \cap \textsc{AAdd}.\ \mathcal{R}\ D_h^+\ \mathbf{x}$$

where the descendent relation D for heap h is defined as:

$$a\ D_h\ \mathbf{y} \iff \mathbf{y} \in addrs(h(a))$$

[5] Strictly speaking, this grammar describes *forests* or ordered lists of abstract trees. We use the term forest when we wish to emphasise the list with a first and last element.

and D_h^+ denotes its transitive closure.

The first two restrictions state that the abstract addresses being used as context holes and the node identifiers are unique across an abstract heap. The last condition states that the abstract addresses join up to produce sensible abstract trees. For instance, $\{\mathbf{x} \to m[\mathbf{y}], \mathbf{y} \to n[\mathbf{x}]\}$ is not an abstract tree heap because of the cycle. An abstract tree heap may be: complete, with no use of abstract addresses; complete, but with the tree split across several heap cells; or *incomplete*, missing some heap cells needed to join some abstract addresses. Incomplete abstract heaps are necessary for local reasoning using the frame rule. In this case, there will be some choice for the missing cells that would render the tree complete.

Given an abstract tree heap h, h^{in} and h^{out} denote the set of abstract cell addresses and context holes, respectively:

$$h^{\mathsf{in}} \triangleq \text{AADD} \cap \text{dom}(h) \qquad h^{\mathsf{out}} \triangleq \text{AADD} \cap (\textstyle\bigcup_{t \in \text{co-dom}(h)} \mathit{addrs}(t)).$$

By design, abstract tree heaps are similar to standard heaps. The construction of a separation algebra over them is thus straightforward.

Definition 2.5 The *separation algebra of abstract tree heaps* is $\mathcal{A}_{\mathbb{T}} \triangleq (H_{\mathbb{T}}, \bullet_{\mathbb{T}}, \mathbf{0}_{\mathbb{T}})$, where $H_{\mathbb{T}}$ is given in Def. 2.4, $\bullet_{\mathbb{T}}$ is the standard disjoint function union with the proviso that the resulting abstract heap is well-formed as per Def. 2.4; and $\mathbf{0}_{\mathbb{T}}$ is a singleton set consisting of the partial function with an empty domain and co-domain.

Since we use variables as resource [14], we pair variable stores for declaring the values of variables with our abstract tree heaps.

Definition 2.6 The *separation algebra of abstract tree states* $\mathcal{SA}_{\mathbb{T}} \triangleq (\Sigma \times H_{\mathbb{T}}, \bullet_{\sigma} \times \bullet_{\mathbb{T}}, \mathbf{0}_{\sigma} \times \mathbf{0}_{\mathbb{T}})$, is the Cartesian product of the separation algebra of variables $(\Sigma, \bullet_{\sigma}, \mathbf{0}_{\sigma})$ [14] and the algebra of abstract tree heaps (Def. 2.5).

To reason about our tree programs, we use the program logic of the Views framework as described in [6]. We instantiate the framework with the separation algebra of abstract tree states (Def. 2.6) as the view monoid and the tree library commands (Def. 2.1) as the atomic commands. What remains is to describe the axioms associated with the tree library commands (Def. 2.7).

Our views are sets of abstract tree states in $\wp(\Sigma \times H_{\mathbb{T}})$. To increase readability of variable sets, we write $\mathsf{var}(\mathtt{x}, v)$ for $\{\mathtt{x} \to v\}$ and $p * q$ for $\{\sigma_1 \bullet_{\sigma} \sigma_2 \mid \sigma_1 \in p \wedge \sigma_2 \in q\}$. For sets of abstract tree heaps, we write $\mathsf{ATree}(d)(\mathbf{x})$ for $\{\mathbf{x} \to d\}$ and $p * q$ for $\{h_1 \bullet_{\mathbb{T}} h_2 \mid h_1 \in p \wedge h_2 \in q\}$. For abstract tree states, we write $\exists v \in V. (p \times q)$ for $\{(\sigma, h) \mid v \in V \wedge \sigma \in p \wedge h \in q\}$ assuming any carrier set V. Finally, we define the *set of complete abstract trees* as the set of abstract trees with no context holes: $\text{COMPDATA}_{\mathbb{T}} = \{d \in \text{DATA}_{\mathbb{T}} \mid \mathit{addrs}(d) = \varnothing\}$.

Definition 2.7 The axiomatisation of atomic tree commands :

$$\text{AXIOM}_{\mathbb{T}} : \text{ATOM}_{\mathbb{T}} \to (\Sigma \times H_{\mathbb{T}}) \times (\Sigma \times H_{\mathbb{T}})$$

is defined in Fig. 3. For soundness, each axiom must preserve the set of abstract addresses present in the described data. Failure to do so would give *unstable* commands, as the abstract connectivity described by abstract addresses would be broken. This is evident in the `deleteTree` axiom, which requires that the subtree

$$\left\{ (\mathsf{var}(\mathtt{n},n) * \mathsf{var}(\mathtt{m},o)) \times (\mathsf{ATree}(n[m[\mathbf{y}] \otimes \mathbf{z}])(\mathbf{x})) \right\}$$
```
m := getFirst(n)
```
$$\left\{ (\mathsf{var}(\mathtt{n},n) * \mathsf{var}(\mathtt{m},m)) \times (\mathsf{ATree}(n[m[\mathbf{y}] \otimes \mathbf{z}])(\mathbf{x})) \right\}$$

$$\left\{ (\mathsf{var}(\mathtt{n},n) * \mathsf{var}(\mathtt{m},o)) \times (\mathsf{ATree}(n[\varnothing])(\mathbf{x})) \right\}$$
```
m := getFirst(n)
```
$$\left\{ (\mathsf{var}(\mathtt{n},n) * \mathsf{var}(\mathtt{m},\mathsf{null})) \times (\mathsf{ATree}(n[\varnothing])(\mathbf{x})) \right\}$$

$$\left\{ (\mathsf{var}(\mathtt{n},n) * \mathsf{var}(\mathtt{m},o)) \times (\mathsf{ATree}(n[\mathbf{y}] \otimes m[\mathbf{z}])(\mathbf{x})) \right\}$$
```
m := getRight(n)
```
$$\left\{ (\mathsf{var}(\mathtt{n},n) * \mathsf{var}(\mathtt{m},m)) \times (\mathsf{ATree}(n[\mathbf{y}] \otimes m[\mathbf{z}])(\mathbf{x})) \right\}$$

$$\left\{ (\mathsf{var}(\mathtt{n},n) * \mathsf{var}(\mathtt{m},o)) \times (\mathsf{ATree}(u[\mathbf{y} \otimes n[\mathbf{z}]])(\mathbf{x})) \right\}$$
```
m := getRight(n)
```
$$\left\{ (\mathsf{var}(\mathtt{n},n) * \mathsf{var}(\mathtt{m},\mathsf{null})) \times (\mathsf{ATree}(u[\mathbf{y} \otimes n[\mathbf{z}]])(\mathbf{x})) \right\}$$

$$\left\{ (\mathsf{var}(\mathtt{n},n) * \mathsf{var}(\mathtt{m},o)) \times (\mathsf{ATree}(m[\mathbf{y} \otimes n[\mathbf{z}] \otimes \mathbf{w}])(\mathbf{x})) \right\}$$
```
m := getUp(n)
```
$$\left\{ (\mathsf{var}(\mathtt{n},n) * \mathsf{var}(\mathtt{m},m)) \times (\mathsf{ATree}(m[\mathbf{y} \otimes n[\mathbf{z}] \otimes \mathbf{w}])(\mathbf{x})) \right\}$$

$$\left\{ (\mathsf{var}(\mathtt{n},n) * \mathsf{var}(\mathtt{m},o)) \times (\mathsf{ATree}(\mathbf{x} \otimes n[\mathbf{y}] \otimes \mathbf{z})(\mathcal{R})) \right\}$$
```
m := getUp(n)
```
$$\left\{ (\mathsf{var}(\mathtt{n},n) * \mathsf{var}(\mathtt{m},\mathsf{null})) \times (\mathsf{ATree}(\mathbf{x} \otimes n[\mathbf{y}] \otimes \mathbf{z})(\mathcal{R})) \right\}$$

$$\{(\mathsf{var}(\mathtt{n},n) * \mathsf{var}(\mathtt{m},o)) \times (\mathsf{ATree}(n[\mathbf{y}])(\mathbf{x}))\}$$
```
m := newNodeAfter(n)
```
$$\left\{ \exists m \in \mathbb{N}^+.\ \begin{array}{l} (\mathsf{var}(\mathtt{n},n) * \mathsf{var}(\mathtt{m},m)) \ \times \\ (\mathsf{ATree}(n[\mathbf{y}] \otimes m[\varnothing])(\mathbf{x})) \end{array} \right\}$$

$$\left\{ (\mathsf{var}(\mathtt{n},n)) \times (\mathsf{ATree}(n[t_c])(\mathbf{x})) \right\}$$
```
deleteTree(n)
```
$$\left\{ (\mathsf{var}(\mathtt{n},n)) \times (\mathsf{ATree}(\varnothing)(\mathbf{x})) \right\}$$

$$\left\{ (\mathsf{var}(\mathtt{m},m) * \mathsf{var}(\mathtt{n},n)) \times (\mathsf{ATree}(m[\mathbf{z}])(\mathbf{y}) * \mathsf{ATree}(n[t_c])(\mathbf{x})) \right\}$$
```
appendChild(m, n)
```
$$\left\{ (\mathsf{var}(\mathtt{m},m) * \mathsf{var}(\mathtt{n},n)) \times (\mathsf{ATree}(m[\mathbf{z} \otimes n[t_c]])(\mathbf{y}) * \mathsf{ATree}(\varnothing)(\mathbf{x})) \right\}$$

Fig. 3. Axiomatisation of tree library commands: assume arbitrary $\mathtt{m}, \mathtt{n} \in \text{PVars}$, $l, m, n, o \in \mathbb{N}^+ \uplus \{\mathsf{null}\}$, $\mathbf{w}, \mathbf{x}, \mathbf{y}, \mathbf{z} \in \text{AAdd}$ and $t_c \in \text{CompData}_{\mathbb{T}}$.

being removed be *complete*, in that it contains no context holes. If the sub-tree did contain a context hole, it would be destroyed, and there would be some matching abstract heap cell which could not be connected anywhere.

The technical details of the Views framework uses a labelled transition system to describe transitions between states. Transitions are labelled either by atomic commands or by id which labels computation steps in which states are not changed. We extend the behaviour of the id transitions of Views by declaring the relation Axiom_{ID} for abstract allocation/deallocation. Abstract (de)allocation does not change the underlying program states and can therefore be seen as id transitions.

Definition 2.8 The *identity axiomatisation*: $\text{Axiom}_{\text{ID}} : (\Sigma \times H_{\mathbb{T}}) \times \{\mathsf{id}\} \times (\Sigma \times H_{\mathbb{T}})$ is given by the abstract allocation and deallocation axioms:

$$\{\mathsf{ATree}(d_1 \circ_{\mathbf{x}} d_2)(a)\} \ \mathsf{id} \ \{\exists \mathbf{y}.\, \mathsf{ATree}(d_1 \circ_{\mathbf{x}} \mathbf{y})(a) * \mathsf{ATree}(d_2)(\mathbf{y})\}$$
$$\{\exists \mathbf{y}.\, \mathsf{ATree}(d_1 \circ_{\mathbf{x}} \mathbf{y})(a) * \mathsf{ATree}(d_2)(\mathbf{y})\} \ \mathsf{id} \ \{\mathsf{ATree}(d_1 \circ_{\mathbf{x}} d_2)(a)\}$$

The existential quantification of the abstract address $\mathbf{y}$ is analogous to the existential quantification used for heap allocation in separation logic.

Definition 2.9 Given the set of abstract tree heaps $H_{\mathbb{T}}$ (Def. 2.4), the set of atomic commands $\text{Atom}_{\mathbb{T}}$ (Def. 2.1) and their axiomatisation $\text{Axiom}_{\mathbb{T}}$ (Def. 2.7), the abstract tree library $\mathbb{T}$ is defined as: $\mathbb{T} \triangleq (H_{\mathbb{T}}, \text{Atom}_{\mathbb{T}}, \text{Axiom}_{\mathbb{T}} \cup \text{Axiom}_{\text{ID}})$

Definition 2.10 Given abstract tree states $p, q \in \wp(\Sigma \times H_{\mathbb{T}})$ (Def. 2.6) and program $C \in \text{Prog}_{\mathbb{T}}$ (Def. 2.1), an *abstract triple* is $\Omega \vDash_{\mathbb{T}} \{p\}C\{q\}$. $\Omega \in \text{PEnv}_{\mathbb{T}}$ is a *procedure specification environment* which is a set of procedure specifications $\mathtt{f} : p_1 \rightarrowtail q_1$, where $\mathtt{f}$ is a procedure name and $p_1, q_1 \in \wp(\Sigma \times H_{\mathbb{T}})$ are pre/post-conditions of $\mathtt{f}$.

3 Concurrent abstract predicates: Tree Implementation

We use concurrent abstract predicates (CAP) to reason about our implementation. We describe the concrete representation of our abstract trees and the concrete implementation of the tree commands in §3.1. We reason about our implementation in §3.2. We give an informal account of how to establish its correctness with respect to the abstract specification in §3.3, and give the formal justification in §4.

3.1 Concrete Tree Implementation

Tree Representation We give a concrete representation of the abstract tree heaps introduced in §2. Consider the abstract tree heap depicted in Fig. 1a. The corresponding concrete tree heap is illustrated in Fig. 4. Each tree node is represented by a node cell with pointers to its left sibling, parent, first child, last child and right sibling where ⊣ denotes a null pointer. With the abstract tree heap, the deletion of the subtree rooted at `n` is a self-contained operation; it does not rely on the context surrounding the subtree. However, this is not the case for the implementation. Since each node cell maintains pointers to its siblings, deletion of the subtree at n requires altering the outgoing pointers of its siblings (and sometimes the parent). In this example, the right pointer of node cell l must be redirected to r and vice-versa; only then can the subtree at n be discarded. Furthermore, in the implementation of the concurrent client program `deleteTree(l) || deleteTree(n)`, both executing threads need to access the pointers between l and n simultaneously, which calls for a suitable synchronisation technique. In our implementation, we synchronise access to these pointers through locking. Each of the left, first, last and right pointers are protected by corresponding locks (lL, dL, eL, rL) that are to be acquired by the competing threads beforehand. We refer to these locks as *pointer locks*.

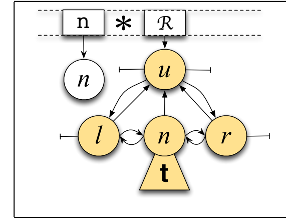

Fig. 4: Concrete tree representation.

We represent each tree node as a node cell consisting of a block of nine consecutive heap cells, $n \to l, u, d, e, r, lL, dL, eL, rL$ where the first five cells contain pointers to the siblings, parent and children, and the last four cells contain the pointer locks. The primary reason for choosing this particular representation was its resemblance to the representation of tree nodes in the Document Object Model (DOM) implementation of the WebKit project [6]. To increase readability, we write $n.l, n.u, \cdots, n.rL$ for $n, n+1, \cdots, n+8$, respectively.

Tree Implementation We provide a concrete implementation of each of the abstract tree commands. Fig. 5 shows our implementation of the `deleteTree` command with some auxiliary procedures. The implementation needs to cater for various cases regarding the siblings of node `n` such as when `n` does not have a left or right sibling. The implementation of the remaining commands are given in the technical report [18]. Note that this implementation is prone to deadlocks; this can occur with our example program `deleteTree(l) || deleteTree(n)` when `l` and `n` are

[6] However, WebKit supports only sequential DOM manipulation and does not use locks.

```
proc deleteTree(n){
  local l,u,d,r  in
    //Acquiring the necessary pointer locks.
1.   u := [n.up] ; lock(n.leftL) ; l:= [n.left] ;
2.   if l ≠ null then lock(l.rightL) else if u ≠ null then lock(u.firstL) ;
3.   lock(n.rightL) ; r:= [n.right] ;
4.   if r ≠ null then lock(r.leftL) ; else if u ≠ null then lock(u.lastL) ;
    //Pointer Swinging.
5.   if l ≠ null then [l.right]:= r else if u ≠ null then [u.first]:= r
6.   if r ≠ null then [r.left]:= l else if u ≠ null then [u.last]:= l
    //Unlocking the acquired pointer locks.
7.  if l ≠ null then unlock(l.rightL) else if u ≠ null then unlock(u.firstL)
8.  if r ≠ null then unlock(r.leftL) else if u ≠ null then unlock(u.lastL)
    //Disposing the sub-tree at node n
9.  d := [n.first] ; call disposeForest(d) ; dealloc(n, 9) ;
}
```

```
proc disposeForest(n){
  local r,d in
    if n ≠ null then
      r := [n.right] ; call disposeForest(r) ;
      d := [n.first] ; call disposeForest(d) ;
      dealloc(n, 9)
}
```

```
proc lock(a){
  while(!CAS(a, 0, 1)) skip ;
}

proc unlock(a){
  [a] := 0 ;
}
```

Fig. 5. Implementation of `deleteTree(n)` and some auxiliary procedures.

adjacent siblings. We focus on this simple implementation here as it is sufficient to describe our ideas. We reason about a *deadlock-free* implementation in [18].

3.2 Reasoning about Concrete Tree Implementation

Recall that the pointers between two sibling nodes (or sometimes a parent and child node) are *shared* resources accessible by both nodes. We use concurrent abstract predicates (CAP) [1] to manage this sharing.

Concurrent Abstract Predicates With CAP, the state is modelled as a pair consisting of a thread-local state and a shared state. The shared state is divided into a set of regions, each encompassing some shared portion of the state. Each region is identified by a region identifier R and is governed by a protocol that describes how the resources of the region can be manipulated. For instance, a lock resource at location x can be specified by:

$$\mathsf{lock}\,(x) \triangleq \exists \mathsf{R}, \pi.\, [\mathcal{L}]^{\mathsf{R}}_{\pi} * \boxed{\mathsf{Unlocked}(\mathsf{R}, x) \,\vee\, \mathsf{Locked}(\mathsf{R}, x)}^{\mathsf{R}}_{T(\mathsf{R},x)}$$

where $\mathsf{Unlocked}(\mathsf{R}, x) \triangleq x \mapsto 0 * [\mathcal{U}]^{\mathsf{R}}_{1}$ and $\mathsf{Locked}(\mathsf{R}, x) \triangleq x \mapsto 1$. This lock definition states that there exists a shared region R containing the lock at location x and the thread's local state contains *some* (for permission $\pi \in (0,1]$) locking capability $[\mathcal{L}]^{\mathsf{R}}_{\pi}$ to acquire it. The region is in one of two states: either the lock is unlocked ($x \mapsto 0$) and the region holds the *full* capability $[\mathcal{U}]^{\mathsf{R}}_{1}$ to unlock it; or the lock is taken ($x \mapsto 1$) and the unlocking capability has been claimed by the locking thread.

The protocol of a shared region is specified through a set of actions, such as actions $\mathcal{L}$ and $\mathcal{U}$ for the lock example. A thread can perform an action if it has a non-zero capability for that action (such as the capability $[\mathcal{L}]^{\mathsf{R}}_{\pi}$ for $\pi > 0$). The

actions permitted on the lock region are declared in $T(\mathsf{R}, x)$:

$$T(\mathsf{R}, x) \triangleq \Big\{ \mathcal{L} : \big(x \mapsto 0 * [\mathcal{U}]_1^{\mathsf{R}}\big) \rightsquigarrow x \mapsto 1 \qquad \mathcal{U} : x \mapsto 1 \rightsquigarrow \big(x \mapsto 0 * [\mathcal{U}]_1^{\mathsf{R}}\big)$$

The action associated with $\mathcal{L}$ changes the state of the shared region from unlocked to locked, moving the capability $[\mathcal{U}]_1^{\mathsf{R}}$ to the thread's local state. The action associated with $\mathcal{U}$ behaves dually, returning $[\mathcal{U}]_1^{\mathsf{R}}$ from the local state to the shared region.

The definition of the CAP separation algebra is given in [1,6]. It provides a set of instrumented states $M_{\mathbb{H}}$ consisting of a local state, a shared state and an action relation capturing the ways in which the shared state can be manipulated. The definition is parametrised by an underlying separation algebra for describing the local state. For this paper, we work with the CAP separation algebra instantiated with the standard fractional heap separation algebra.

Definition 3.1 The *CAP separation algebra*, instantiated with fractional heap separation algebra, is $\mathcal{A}_{\mathbb{C}_{\mathbb{H}}} \triangleq (M_{\mathbb{H}}, \bullet_{\mathbb{C}_{\mathbb{H}}}, \mathbf{0}_{\mathbb{C}_{\mathbb{H}}})$. The *separation algebra of CAP states* is $\mathcal{SA}_{\mathbb{C}_{\mathbb{H}}} \triangleq (\Sigma \times M_{\mathbb{H}}, \bullet_\sigma \times \bullet_{\mathbb{C}_{\mathbb{H}}}, \mathbf{0}_\sigma \times \mathbf{0}_{\mathbb{C}_{\mathbb{H}}})$, where $\times$ denotes standard Cartesian product.

Recall that we base our reasoning on the Views framework [6] which provides general reasoning about a generic programming language parametrised by a set of atomic commands and their axiomatisation.

Definition 3.2 Let $\textsc{Atom}_{\mathbb{H}}$ be the set of atomic heap commands including assignment, value lookup, memory allocation/deallocation and the compare and set construct `CAS`. We write $\textsc{Prog}_{\mathbb{H}}$ for the set of programs generated from Views' programming language instantiated with set $\textsc{Atom}_{\mathbb{H}}$. Let $\textsc{Axiom}_{\mathbb{H}}$ be the standard set of separation-logic axiomatisations for these commands based on the fractional heap separation algebra. The Views framework provides the associated reasoning for $\textsc{Prog}_{\mathbb{H}}$.

Definition 3.3 Given the CAP separation algebra $\mathcal{A}_{\mathbb{C}_{\mathbb{H}}}$ (Def. 3.1), the atomic heap commands $\textsc{Atom}_{\mathbb{H}}$ and their axiomatisation $\textsc{Axiom}_{\mathbb{H}}$, the *CAP library* $\mathbb{C}_{\mathbb{H}}$ for fractional heaps is defined as: $\mathbb{C}_{\mathbb{H}} \triangleq (\mathcal{A}_{\mathbb{C}_{\mathbb{H}}}, \textsc{Atom}_{\mathbb{H}}, \textsc{Axiom}_{\mathbb{H}})$.

Definition 3.4 Given CAP states $p, q \in \wp(\Sigma \times M_{\mathbb{H}})$ (Def. 3.1) and program $C \in \textsc{Prog}_{\mathbb{H}}$ (Def. 3.2), a *concrete triple* is $\Omega \vDash_{\mathbb{C}_{\mathbb{H}}} \{p\}C\{q\}$. $\Omega \in \mathrm{PE}\textsc{nv}_{\mathbb{C}_{\mathbb{H}}}$ is a set of procedure specifications $\mathtt{f} : p_1 \rightarrowtail q_1$, where $\mathtt{f}$ is a procedure name and $p_1, q_1 \in \wp(\Sigma \times M_{\mathbb{H}})$ are pre/post-conditions of $\mathtt{f}$.

In §4, we define the library refinement $\tau : \mathbb{T} \to \mathbb{C}_{\mathbb{H}}$ for abstract tree library $\mathbb{T}$ (Def. 2.9) and concrete CAP library $\mathbb{C}_{\mathbb{H}}$ (Def. 3.3). As a first step in defining τ, we identify the concrete CAP states corresponding to the abstract tree fragments. These concrete CAP states rely on an *interface function* I_τ linking abstract addresses with concrete pointers.

Interface Functions. Fig. 4 illustrates a concrete heap representation of a complete abstract tree at root address $\mathcal{R}$. However, many of the abstract tree commands (e.g. `deleteTree`) are specified using tree fragments rather than complete trees. We need to understand the concrete representation of tree fragments.

When an abstract tree heap is split using abstract allocation, the constituent heaps are agnostic to one-another's shapes. For instance, consider the abstract tree heap $\mathcal{R} \mapsto l \otimes i \otimes j \otimes r$ which can be split as $h_1 \bullet_{\mathbb{T}} h_2$ where $h_1 \triangleq \mathcal{R} \mapsto l \otimes \mathbf{x} \otimes r$ and $h_2 \triangleq \mathbf{x} \mapsto i \otimes j$. The abstract tree heap h_1 embodies no knowledge of the tree placed within $\mathbf{x}$; *mutatis mutandis* for h_2. At the concrete level, the situation is different. The concrete representation of h_2 relies on additional knowledge from the concrete representation of h_1, since the representation of node i includes pointers to its left sibling (l) and parent (u). Thus, when translating an abstract heap with abstract addresses, we require supplementary information originating from the concrete heap. We track this additional piece of information associated with each abstract address $\mathbf{x} \in$ AAdd through a concrete *interface*. Consider Fig. 6, which depicts an abstract tree at abstract cell $\mathbf{x}$ (left) and its concrete representation (right) assuming an *interface function* $I_\mathcal{T}$. In the concrete representation, the solid lines represent resources held locally, such as node n, its up pointer and the subtree t, while the dashed lines denote shared resources, such as the pointers between node n and its siblings. A concrete *in-interface* records the address of the first (i) and last (j) nodes of the concrete representation of the forest pointed to by $\mathbf{x}$; in our example, the in-interface is $in \triangleq (n, n)$. A concrete *out-interface* records the addresses of the parent node (u) and nodes placed immediately to the left (l) and right (r) of the abstract address $\mathbf{x}$; in our example, the out-interface is $out \triangleq (l^x, u^x, r^x)$. The interface function $I_\mathcal{T}$ maps $\mathbf{x}$ to (in, out).

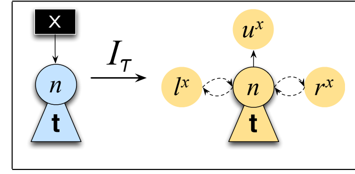

Fig. 6: Tree Fragments.

Definition 3.5 The sets of concrete *in-* and *out-interfaces* are defined by:

$$\text{In}_\mathcal{T} \triangleq (\mathbb{N}^+ \uplus \{\mathsf{null}\})^2 \qquad\qquad \text{Out}_\mathcal{T} \triangleq (\mathbb{N}^+ \uplus \{\mathsf{null}\})^3$$

The set of *in-* and *out-interface functions* are defined by $\mathcal{I}^{\mathsf{in}}_\mathcal{T} \triangleq \wp(\text{AAdd} \rightharpoonup \text{In}_\mathcal{T})$ and $\mathcal{I}^{\mathsf{out}}_\mathcal{T} \triangleq \wp(\text{AAdd} \rightharpoonup \text{Out}_\mathcal{T})$. The set of interface functions is $\mathcal{I}_\mathcal{T} \triangleq \mathcal{I}^{\mathsf{in}}_\mathcal{T} \times \mathcal{I}^{\mathsf{out}}_\mathcal{T}$.

CAP Representation of Trees Fig. 7 defines concurrent abstract predicates for describing the concrete representation of abstract tree heaps, parameterised by interface function $I_\mathcal{T}$. The $\mathsf{CTree}^{I_\mathcal{T}}(t)(\mathcal{R})$ predicate provides a concrete representation of the tree fragment t at root address $\mathcal{R}$. The predicate $\mathsf{CTree}^{I_\mathcal{T}}(t)(\mathbf{x})$ provides a concrete representation of the tree fragment at abstract address $\mathbf{x}$. Both predicates are defined using a ICTree predicate indexed by in- and out-interfaces determined by $I_\mathcal{T}$. For $\mathcal{R}$, the out-interface is $(\mathsf{null}, \mathsf{null}, \mathsf{null})$. The in-interface is unknown at this point and hence is existentially quantified. Its value is later found using the ICTree predicate. The interface of $\mathbf{x}$ is determined by the interface function $I_\mathcal{T}$.

We now describe the ICTree predicate with the explicit in- and out-interface arguments (i, j) and (l, u, r), following the cases of the inductive definition.

$\mathbf{ICTree}^{I_\mathcal{T}}(\varnothing)$ There is no resource in the concrete heap representation, simply some pointer equalities. The assertion $i \doteq r$ simply states that the address of the first node cell in the forest is equal to the address of the node cell immediately to the right of the tree; similarly for j and l. This is to ensure the correct concrete representation

$$
\begin{aligned}
\mathsf{CTree}^{I_\tau}(t)(\mathcal{R}) &\triangleq \exists i,j.\, \mathsf{ICTree}^{I_\tau}\,(t)\,(i,j)\,(\mathsf{null},\mathsf{null},\mathsf{null})\\
\mathsf{CTree}^{I_\tau}(t)(\mathbf{x}) &\triangleq \mathsf{ICTree}^{I_\tau}\,(t)\,(i^x,j^x)\,(l^x,u^x,r^x)\\
&\quad * {I_\tau}^{\mathsf{in}}(\mathbf{x}) \doteq (i^x,j^x) * {I_\tau}^{\mathsf{out}}(\mathbf{x}) \doteq (l^x,u^x,r^x)\\
\mathsf{ICTree}^{I_\tau}\,(\varnothing)\,(i,j)\,(l,u,r) &\triangleq (i \doteq r) * (j \doteq l)\\
\mathsf{ICTree}^{I_\tau}\,(\mathbf{x})\,(i,j)\,(l,u,r) &\triangleq {I_\tau}^{\mathsf{in}}(\mathbf{x}) \doteq (i,j) * {I_\tau}^{\mathsf{out}}(\mathbf{x}) \doteq (l,u,r)\\
\mathsf{ICTree}^{I_\tau}\,(t_1 \otimes t_2)\,(i,j)\,(l,u,r) &\triangleq \exists p,q.\ \mathsf{ICTree}^{I_\tau}\,(t_1)\,(i,p)\,(l,u,q)\\
&\quad * \mathsf{ICTree}^{I_\tau}\,(t_2)\,(q,j)\,(p,u,r)\\
\mathsf{ICTree}^{I_\tau}\,(n[t])\,(i,j)\,(l,u,r) &\triangleq i \doteq j \doteq n \ *\ n.u \mapsto u\ * \exists d,e.\\
&\quad \left(\begin{array}{l} \mathsf{Left}\,(n,l,u) * \mathsf{First}\,(n,d) * \mathsf{Last}\,(n,e) * \mathsf{Right}\,(n,r,u)\\ * \mathsf{ICTree}^{I_\tau}\,(t)\,(d,e)\,(\mathsf{null},n,\mathsf{null}) \end{array}\right)
\end{aligned}
$$

Fig. 7. CAP Representation of Abstract Tree Heaps: we write $a \doteq b$ for $a = b \wedge \mathsf{emp}$.

of trees such as $t_1 \otimes \varnothing \otimes t_2$.

$\mathsf{ICTree}^{I_\tau}(\mathbf{x})$ There is no resource in the concrete heap representation, simply the appropriate connection between abstract address $\mathbf{x}$ and the interfaces given by I_τ.

$\mathsf{ICTree}^{I_\tau}(t_1 \otimes t_2)$ This is simply the $*$-composition of the concrete representations of the constituent abstract subtrees, given an appropriate choice of interfaces.

$\mathsf{ICTree}^{I_\tau}(n[t])$ The first two lines provide the concrete representation of the node n, and the last line represents the subtree t. For the subtree, the concrete representation is straightforward. For the node, $i \doteq j \doteq n$ since there is only one tree in the forest. The concrete representation has full ownership of the parent pointer $n.u \mapsto u$, as there is no competition for this resource from other nodes. The Left, Right, First and Last predicates represent the pointers and locks associated with the left sibling, right sibling, first child and last child, which are in competition with the representations of nearby nodes and hence must be shared.

We give the definition of Left predicate here; the Right, First and Last predicates are defined analogously and we defer them to the accompanying technical report [18].

The definition of the Left predicate is given in Fig. 8 and is described by the isLLock and ownsR predicates.

$\mathsf{isLLock}(n,l,u,0.5)$ This predicate states that there exists a shared region R_{nl} containing the left pointer lock of node n; the thread's local state contains some locking capability $[\mathcal{L}]_{0.5}^{\mathsf{R}_{nl}}$ associated with the region.

$\mathsf{ownsR}(n,l,u,1)$ Similar to isLLock, this predicate refers to the R_{nl} region where the thread's local state holds *full* permission on the *witness* capability $[\mathcal{W}]_1^{\mathsf{R}_{nl}}$. This is to denote that even though the pointers between n and its left sibling l are shared, node n itself is owned by the current thread. Later in the description of LWit predicate we demonstrate how this capability can be used to track the identity of the thread that has claimed the left pointer lock of n.

The contents of the R_{nl} region are analogous to that of the lock example given

$$\mathsf{Left}(n,l,u) \triangleq \mathsf{isLLock}(n,l,u,0.5) * \mathsf{ownsR}(n,l,u,1)$$

$$\mathsf{isLLock}(n,l,u,\pi) \triangleq \exists \mathsf{R}_{nl}.\, [\mathcal{L}]_{\pi}^{\mathsf{R}_{nl}} * \boxed{\begin{array}{l}\mathsf{LUnlocked}(\mathsf{R}_{nl},n,l,u)\\ \vee\, \mathsf{LLocked}(\mathsf{R}_{nl},n,l,u)\end{array}}^{\mathsf{R}_{nl}}_{\mathsf{LC}(\mathsf{R}_{nl},n,u)}$$

$$\mathsf{ownsR}(n,l,u,\pi) \triangleq \exists \mathsf{R}_{nl}.\, [\mathcal{W}]_{\pi}^{\mathsf{R}_{nl}} * \boxed{\begin{array}{l}\mathsf{LUnlocked}(\mathsf{R}_{nl},n,l,u)\\ \vee\, \mathsf{LLocked}(\mathsf{R}_{nl},n,l,u)\end{array}}^{\mathsf{R}_{nl}}_{\mathsf{LC}(\mathsf{R}_{nl},n,u)}$$

$$\mathsf{LUnlocked}(\mathsf{R}_{nl},n,l,u) \triangleq n.lL \to 0 * [\mathcal{U}]_{1}^{\mathsf{R}_{nl}} * \Subset^{n,l,u}$$

$$\Subset^{n,l,u} \triangleq n.l \overset{0.5}{\mapsto} l * \left(\begin{array}{l}\left(l \neq \mathsf{null} \wedge l.r \overset{0.5}{\mapsto} n * \mathsf{isRLock}(l,n,u,0.5)\right)\\ \vee \left(l = \mathsf{null} \wedge u \neq \mathsf{null} \wedge u.d \overset{0.5}{\mapsto} n * \mathsf{isDLock}(u,n,0.5)\right)\\ \vee \left(l \doteq u \doteq \mathsf{null} * n.l \overset{0.5}{\mapsto} l * \mathsf{isLLock}(n,l,u,0.5)\right)\end{array}\right)$$

$$\mathsf{LLocked}(\mathsf{R}_{nl},n,l,u) \triangleq n.lL \to 1 * \mathsf{LWit}(l,n,u)$$

$$\begin{array}{rl}\mathsf{LWit}(l,n,u) \triangleq & \mathsf{ownsR}(l,n,u,0.5)\\ & \vee\ (l \neq \mathsf{null} \wedge \mathsf{ownsL}(l,n,u,0.5))\\ & \vee (l = \mathsf{null} \wedge u \neq \mathsf{null} \wedge \mathsf{ownsD}(u,n,0.5))\end{array}$$

$$\mathsf{LC}(\mathsf{R}_{nl},n,u) \triangleq \begin{cases}\mathcal{L}: & \mathsf{LUnlocked}(\mathsf{R}_{nl},n,l,u) \rightsquigarrow \mathsf{LLocked}(\mathsf{R}_{nl},n,l,u)\\ \mathcal{U}: & \mathsf{LLocked}(\mathsf{R}_{nl},n,l,u) \rightsquigarrow \mathsf{LUnlocked}(\mathsf{R}_{nl},n,l',u)\\ \mathcal{W}: & \mathsf{LInit}(n,l,u) \rightsquigarrow \mathsf{LUnlocked}(\mathsf{R}_{nl},n,l,u) \vee \mathsf{LLocked}(\mathsf{R}_{nl},n,l,u)\\ & \mathsf{LUnlocked}(\mathsf{R}_{nl},n,l,u) \vee \mathsf{LLocked}(\mathsf{R}_{nl},n,l,u) \rightsquigarrow \mathsf{LInit}(n,l,u)\end{cases}$$

Fig. 8. The definition of Left and its auxiliary predicates.

before. The difference is the additional pointer resource ($\Subset$) contained within the region. The region can be in one of two states: LUnlocked or LLocked.

LUnlocked$(\mathsf{R}_{nl}, n, l, u)$ In this state, the left pointer lock is not taken ($n.lL \to 0$), and the full unlocking capability ($[\mathcal{U}]_{1}^{\mathsf{R}_{nl}}$) and the $\Subset$ resource are in the region.

$\Subset^{n,l,u}$ This predicate describes the pointer resources between node n and its left hand side depending on its position in the tree. When n has a left sibling ($l \neq \mathsf{null}$), it consists of partial ownership of the pointers between n and l, that is, $n.l \overset{0.5}{\mapsto} l * l.r \overset{0.5}{\mapsto} n$. It also contains the capability to acquire the right pointer lock of l, in order to obtain full ownership of the pointers between node n and l. This is captured by the $\mathsf{isRLock}(l,n,u,0.5)$ predicate. On the other hand, if n does not have a left sibling and is thus the first child of node u ($l = \mathsf{null} \wedge u \neq \mathsf{null}$), the $\Subset$ resource consists of partial ownership on the left pointer of n and the first pointer of u ($n.l \overset{0.5}{\mapsto} l * u.d \overset{0.5}{\mapsto} n$) as well as the capability to acquire the first pointer lock of u ($\mathsf{isDLock}(u,n,0.5)$). Finally, if n is the first child underneath the root address

$\mathcal{R}$, and thus both its left sibling and parent correspond to null ($l = u =$ null), the $\Subset$ resource consists of full permission on n's left pointer ($n.l \overset{1}{\mapsto} l$) and the remaining locking capability on region R_{nl} ($\mathsf{isLLock}\,(n, l, u, 0.5)$).

LLocked$(\mathsf{R}_{nl}, n, l, u)$ In this state, the left pointer lock is taken ($n.lL \mapsto 1$), and the capability $[\mathcal{U}]_1^{\mathsf{R}_{nl}}$ and the pointer resources $\Subset$ have been claimed by the locking thread in exchange for the LWit resource.

LWit(l, n, u) Since the left pointer lock of n can only be be acquired by the thread in possession of node n, *or* node l (or node u if n is the first child of u and does not have a left sibling), the LWit predicate is used to track the identity of the locking thread. Recall from the definition of the Left predicate that the full *witness* capability $[\mathcal{W}]_1^{\mathsf{R}_{nl}}$ is held by the thread that owns node n, as described by $\mathsf{ownsR}\,(n, l, u, 1)$. We thus use the witness capabilities to determine the identity of the locking thread. The first disjunct denotes the case where the lock has been claimed by the thread in possession of node n and corresponds to the witness capability on this region ($\mathsf{ownsR}\,(n, l, u, 0.5)$). Analogously, the second disjunct represents the case where the thread in possession of node l has acquired the lock, denoted by ($\mathsf{ownsR}\,(n, l, u, 0.5)$). The third disjunct captures the case where n does not have a left sibling and the lock has been taken by the thread that owns the parent node u ($\mathsf{ownsD}\,(u, n, 0.5)$).

LC(R_{nl}, n, u) The R_{nl} region is governed by the $\mathsf{LC}(\mathsf{R}_{nl}, n, u)$ protocol describing the ways in which its contents can be manipulated through actions. The action associated with $\mathcal{L}$ changes the state of R_n from LUnlocked to LLocked. Dually, the action of $\mathcal{U}$ changes the state of the region from LLocked to LUnlocked. The first action of $\mathcal{W}$ *initialises* the contents of R_{nl} immediately after its creation. Similarly, the second action *finalises* the contents of the region right before its destruction. The initial/final contents of the region are realised by the LInit predicate. The definition of LInit predicate is nonessential as it bears no relevance in understanding the interactions between the threads and the shared region and is thus omitted here.

Recall that abstract trees can be split and joined using abstract allocation and deallocation. We show that concrete trees can be split (joined) analogously provided that the interface function is extended (reduced) accordingly to capture the interface associated with the freshly allocated (deallocated) abstract address.

Theorem 3.6 (Abstract (de)allocation) *For all interface functions* $I_\tau \in \mathcal{I}_\tau$*, addresses* $a \in \textsc{Add}_{\mathbb{T}}$ *and trees* $t_1, t_2 \in \textsc{Data}_{\mathbb{T}}$*:*

$$\mathit{CTree}^{I_\tau}(t_1 \circ_{\mathbf{x}} t_2)(a) \equiv \exists \mathbf{y} \in \textsc{AAdd}, in \in \textsc{In}_\tau, out \in \textsc{Out}_\tau. \\ \mathit{CTree}^{I'_\tau}(t_1 \circ_{\mathbf{x}} \mathbf{y})(a) * \mathit{CTree}^{I'_\tau}(t_2)(\mathbf{y})$$

where $\textsc{In}_\tau, \textsc{Out}_\tau$ *and* $\mathcal{I}_\tau$ *are given in Def. 3.5 and* $I'_\tau \triangleq I_\tau \uplus ([\mathbf{y} \mapsto in], [\mathbf{y} \mapsto out])$.

Proof. The proof of this theorem is provided in the technical report [18].

3.3 Soundness of Concrete Tree Library $\mathbb{C}_{\mathbb{H}}$

In order to show that our concrete CAP library $\mathbb{C}_{\mathbb{H}}$ (Def. 3.3) is sound with respect to the abstract tree library $\mathbb{T}$ (Def. 2.9), we show that everything that can be proved about the abstract library $\mathbb{T}$, can also be proved about the concrete library $\mathbb{C}_{\mathbb{H}}$. That is, for every abstract triple $\Omega \vDash_{\mathbb{T}} \{p\}C\{q\}$ (Def. 2.10), we show that there exists a corresponding concrete triple $\Omega' \vDash_{\mathbb{C}_{\mathbb{H}}} \{p'\}C'\{q'\}$ (Def. 3.4) where C' is the implementation of C and p', q' denote the concrete representations of p, q, respectively. For instance, we need to show that the implementation of the `deleteTree` command in Fig. 5, satisfies its abstract specification as given in Fig. 3.

In §4, we formalise triple transformation (Def. 4.6) and define what it means for the CAP library $\mathbb{C}_{\mathbb{H}}$ to be sound with respect to the abstract tree library $\mathbb{T}$ (Def. 4.8). We then prove that $\mathbb{C}_{\mathbb{H}}$ is sound with respect to $\mathbb{T}$ (Theorem 4.9). In what follows, we give an informal account of establishing the correctness of the `deleteTree` implementation and state a *pseudo-theorem* that *almost* captures the desired result. We defer the formalisation of the correct theorem to §4 (Theorem 4.7), and the full proof to the accompanying technical report [18].

Pseudo-Theorem 1 *Let* $[\![\texttt{deleteTree(n)}]\!]_{\tau}$ *denote the implementation of* `deleteTree(n)` *command in Fig. 5. The following statement* almost *holds:*

$$\forall I_c \in \mathcal{I}_{\tau}. \quad \left\{ \exists I_d \in \mathcal{I}_{\tau}^{\texttt{del}}. \left(\mathit{var}(\texttt{n}, n) \times \mathit{CTree}^{I_c \uplus I_d}(n[t])(\mathbf{x}) \right) \right\}$$
$$[\![\texttt{deleteTree(n)}]\!]_{\tau}$$
$$\left\{ \exists I_d \in \mathcal{I}_{\tau}^{\texttt{del}}. \left(\mathit{var}(\texttt{n}, n) \times \mathit{CTree}^{I_c \uplus I_d}(\varnothing)(\mathbf{x}) \right) \right\}$$

with $\qquad \mathcal{I}_{\tau}^{\texttt{del}} \triangleq \left\{ I \in \mathcal{I}_{\tau} \mid \mathit{dom}(I^{in}) = \{\mathbf{x}\} \wedge \mathit{dom}(I^{out}) = \varnothing \right\}.$

The pre-condition contains the variable resource $\mathsf{var}(\texttt{n}, n)$ and the concrete tree representation $\mathsf{CTree}(n[t])(\mathbf{x})$ of complete subtree at abstract address $\mathbf{x}$. The pre-condition gives the right to control and modify the *inner* interface $I_{\tau}^{\texttt{del}}$ of address $\mathbf{x}$ (in this case, (n, n)). This freedom to modify $I_{\tau}^{\texttt{del}}$ is captured by the existential quantification of I_d in both pre- and post-conditions. Note that the control over the *outer* interface of $\mathbf{x}$ as well as the interfaces associated with other abstract addresses lies with the context. We therefore need to be agnostic to the interfaces associated with abstract addresses outside the domain of I_d and show that the implementation is correct for *all* valid choices of context interfaces I_c. This is realised by the universal quantification of I_c outside the Hoare triple. However, since we are reasoning about *concurrent* programs, during the execution of `deleteTree`, the interfaces captured by I_c are potentially changing underfoot. Hence, contrary to what the above triple states, the value of I_c in the pre-condition does not necessarily agree with that of I_c in the post-condition. The correct theorem is given in §4.

4 Correct Library Translation

We state what it means to correctly implement our abstract tree library $\mathbb{T}$ (Def. 2.9) with our CAP library $\mathbb{C}_{\mathbb{H}}$ (Def. 3.3) by defining a *library translation* $\tau : \mathbb{T} \to \mathbb{C}_{\mathbb{H}}$. In §4.1, we give our specific library translation τ and, in §4.2, we prove its soundness.

In the accompanying technical report [18], we generalise this approach to translate arbitrary abstract libraries, and stipulate a set of general properties that when satisfied, will warrant sound translation of any abstract library. These properties are stated for our specific library translation in Theorem 4.9.

4.1 Tree to CAP Translation

Our goal is to define a translation $\tau : \mathbb{T} \to \mathbb{C}_{\mathbb{H}}$ in such a way that would allow us to *correctly* transform abstract tree triples to concrete CAP triples, following the spirit of Pseudo-theorem 1. We give an *abstract tree heap translation* that maps abstract tree heaps to CAP heaps. To keep the translation concise, we define it using CAP assertions which can then be interpreted as elements of $\wp(M_{\mathbb{H}})$ [1].

Definition 4.1 Given the separation algebra of abstract tree heaps (Def. 2.5) $\mathcal{A}_{\mathbb{T}} \triangleq (H_{\mathbb{T}}, \bullet_{\mathbb{T}}, \mathbf{0}_{\mathbb{T}})$, the tree interface function $\mathcal{I}_\tau$ (Def. 3.5) and the CAP separation algebra (Def. 3.1) $\mathcal{A}_{\mathbb{C}_{\mathbb{H}}} \triangleq (M_{\mathbb{H}}, \bullet_{\mathbb{C}_{\mathbb{H}}}, \mathbf{0}_{\mathbb{C}_{\mathbb{H}}})$, the *abstract tree heap translation* $\langle . \rangle_\tau^{(\cdot)} : \mathcal{H}_{\mathbb{T}} \to \mathcal{I}_\tau \to \wp(M_{\mathbb{H}})$ is a function defined inductively over the structure of abstract tree heaps:

$$\langle \mathbf{0}_{\mathbb{T}} \rangle_\tau^I \triangleq \mathrm{emp} \qquad \langle \mathcal{R} \to t \rangle_\tau^I \triangleq \mathsf{CTree}^I(t)(\mathcal{R}) \qquad \langle \mathbf{x} \to t \rangle_\tau^I \triangleq \mathsf{CTree}^I(t)(\mathbf{x})$$
$$\langle h_1 \bullet_{\mathbb{T}} h_2 \rangle_\tau^I \triangleq \exists I_1, I_2.\, I = I_1 \cup I_2 \wedge \langle h_1 \rangle_\tau^{I_1} * \langle h_2 \rangle_\tau^{I_2}$$

where the definition of the CTree predicate is as given in Fig. 7.

We also need to transform programs in $\textsc{Prog}_{\mathbb{T}}$ to programs in $\textsc{Prog}_{\mathbb{H}}$.

Definition 4.2 The *implementation function* $[\![.]\!]_\tau : \textsc{Prog}_{\mathbb{T}} \to \textsc{Prog}_{\mathbb{H}}$, replaces each call to an atomic tree command in $\textsc{Atom}_{\mathbb{T}}$ with a call to a correspondingly named procedure denoting its implementation, while other program constructs remain unchanged. The implementation of `deleteTree` is as given in Fig. 5; the implementation of other atomic commands are given in the technical report [18].

Definition 4.3 The translation $\tau : \mathbb{T} \to \mathbb{C}_{\mathbb{H}}$ is a triple comprising the set of interfaces $\textsc{In}_\tau \times \textsc{Out}_\tau$ (Def. 3.5), the abstract tree heap translation $\langle . \rangle_\tau^{(\cdot)}$ (Def. 4.1) and the implementation function $[\![.]\!]_\tau$ (Def. 4.2): $\tau \triangleq \left(\textsc{In}_\tau \times \textsc{Out}_\tau, \langle . \rangle_\tau^{(\cdot)}, [\![.]\!]_\tau\right)$.

4.2 Correctness of Translation

Given the translation τ, our goal is to state and prove the correctness of a translation of abstract tree triples $\Omega \vDash_{\mathbb{T}} \{p\}C\{q\}$ to concrete CAP triples $\Omega' \vDash_{\mathbb{C}_{\mathbb{H}}} \{p'\}\, [\![C]\!]_\tau \,\{q'\}$ in the spirit of Pseudo-Theorem 1. To do this, we need to define a *state translation* between abstract tree states and concrete CAP states.

Recall from §3.3 that the universal quantification of context interface function I_c in Pseudo-Theorem 1 is incorrect, since interfaces captured by I_c are subject to change by the environment. We define the set of *stable interface functions* $\mathcal{SI}_\tau$ whereby an abstract address is associated not with a single interface but with a *set* of interfaces. By associating a set of possible interfaces with an abstract address at any one time, we can account for the potential change of interfaces by the environment.

Definition 4.4 Given the sets of inner and outer interfaces $\textsc{In}_\tau$, $\textsc{Out}_\tau$ (def. 3.5), the *set of stable inner-interface functions* is $\mathcal{SI}_\tau{}^{\mathsf{in}} : \wp(\textsc{AAdd} \rightharpoonup \wp(\textsc{In}_\tau))$ and the *set of stable outer-interface functions* is $\mathcal{SI}_\tau{}^{\mathsf{out}} : \wp(\textsc{AAdd} \rightharpoonup \wp(\textsc{Out}_\tau))$. The set of *stable interface functions* is defined by $\mathcal{SI}_\tau \triangleq \mathcal{SI}_\tau{}^{\mathsf{in}} \times \mathcal{SI}_\tau{}^{\mathsf{out}}$. For $SI \in \mathcal{SI}_\tau$, SI^{in} and SI^{out} denote the first and second projections, respectively. The *collapse* of SI is $SI{\downarrow} \triangleq \left(SI{\downarrow}^{\mathsf{in}} \times SI{\downarrow}^{\mathsf{out}}\right)$ where $SI^{\mathsf{in}}{\downarrow}$ (and analogously $SI{\downarrow}^{\mathsf{out}}$) is defined as:

$$SI^{\mathsf{in}}{\downarrow} \triangleq \left\{ I \in \mathcal{I}_\tau^{\mathsf{in}} \;\middle|\; \mathrm{dom}(I) = \mathrm{dom}(SI^{\mathsf{in}}) \;\wedge\; \forall \mathbf{x} \in \mathrm{dom}(I).\, I(\mathbf{x}) \in SI^{\mathsf{in}}(\mathbf{x}) \right\}$$

We can now give a state translation between abstract tree states and concrete CAP states, parametrised by the stable interface functions.

Definition 4.5 The *state translation*: $\langle\!|.|\!\rangle_\tau^{(.)} : \wp(\Sigma \times H_{\mathbb{T}}) \to \mathcal{SI}_\tau \to \wp(\Sigma \times M_{\mathbb{H}})$ is:

$$\langle\!|p|\!\rangle_\tau^{SI} \triangleq \left\{ (\sigma, m) \;\middle|\; \begin{array}{l} (\sigma, h) \in p \;\wedge \\ \exists I_d.\, m \in \bigcup\limits_{I_c \in SI\downarrow} \left\{ \langle h \rangle_\tau^{I_c \uplus I_d} \;\middle|\; \mathrm{dom}(I_d{}^{\mathsf{in}}) = h^{\mathsf{in}} \;\wedge\; \mathrm{dom}(I_d{}^{\mathsf{out}}) = h^{\mathsf{out}} \right\} \end{array} \right\}$$

where h^{in}, h^{out} are given in Def. 2.4 and $I_1 \uplus I_2 \triangleq \left(I_1{}^{\mathsf{in}} \uplus I_2{}^{\mathsf{in}},\; I_1{}^{\mathsf{out}} \uplus I_2{}^{\mathsf{out}}\right)$ with $\uplus$ denoting the standard disjoint function union.

Note that when transforming a tree state, the variable store σ remains unchanged while the tree heap h is translated as $\langle h \rangle_\tau^{I_c \uplus I_d}$. I_d captures the interfaces of abstract addresses within the control of the thread and thus are in the footprint of h. On the other hand, I_c represents the interfaces of abstract addresses controlled by the environment. I_d and I_c are analogous to those of Pseudo-theorem 1, with the exception that they are drawn from the *stable interface function* SI.

We can now formalise the transformation of an abstract triple $\Omega \vDash_{\mathbb{T}} \{p\}C\{q\}$ to a concrete triple $(\!|\Omega|\!)_\tau \vDash_{\mathbb{C}_{\mathbb{H}}} \{\langle\!|p|\!\rangle_\tau\} [\![C]\!]_\tau \{\langle\!|q|\!\rangle_\tau\}$, where $(\!|.|\!)_\tau$ is defined in Def. 4.6.

Definition 4.6 Given the library translation $\tau : \mathbb{T} \to \mathbb{C}_{\mathbb{H}}$, for all procedure specification environments $\Omega \in \textsc{PEnv}_{\mathbb{T}}$, abstract tree states $p, q \in \wp(\Sigma \times H_{\mathbb{T}})$ and tree programs $C \in \textsc{Prog}_{\mathbb{T}}$, the *translated triple* $\tau :\; \Omega \vDash_{\mathbb{T}} \{p\}C\{q\}$ is:

$$\tau :\; \Omega \vDash_{\mathbb{T}} \{p\}\; C\; \{q\} \;\triangleq\; \forall SI \in \mathcal{SI}_\tau.\;\; (\!|\Omega|\!)_\tau \vDash_{\mathbb{C}_{\mathbb{H}}} \left\{\langle\!|p|\!\rangle_\tau^{SI}\right\}\; [\![C]\!]_\tau\; \left\{\langle\!|q|\!\rangle_\tau^{SI}\right\}$$

where $(\!|\Omega|\!)_\tau \triangleq \{\mathtt{f} : \langle\!|p|\!\rangle_\tau^{SI} \rightarrowtail \langle\!|q|\!\rangle_\tau^{SI} \mid (\mathtt{f} : p \rightarrowtail q) \in \Omega \;\wedge\; SI \in \mathcal{SI}_\tau\}$.

We can now amend the statement of Pseudo-theorem 1 and formulate the theorem describing the correctness of the `deleteTree` command.

Theorem 4.7 (`deleteTree` Correctness) *The implementation of* `deleteTree` *command as given in Fig. 5 is correct if*

$$\tau :\; \Omega \vDash_{\mathbb{T}} \left\{ \mathsf{var}(\mathtt{n}, n) \times \mathsf{ATree}(n[t])(\mathbf{x}) \right\} \mathtt{deleteTree} \left\{ \mathsf{var}(\mathtt{n}, n) \times \mathsf{ATree}(\varnothing)(\mathbf{x}) \right\}$$

Proof. *The correctness proof of the above statement is given in the accompanying technical report [18].*

Definition 4.8 The library translation $\tau : \mathbb{T} \to \mathbb{C}_{\mathbb{H}}$ is *sound* if, for all $\Omega \in \textsc{PEnv}_{\mathbb{T}}$, $p, q \in \wp(\Sigma \times H_{\mathbb{T}})$ and $C \in \textsc{Prog}_{\mathbb{T}}$: $\quad \Omega \vDash_{\mathbb{T}} \{p\}\; C\; \{q\} \implies \tau :\; \Omega \vDash_{\mathbb{T}} \{p\}\; C\; \{q\}$

Theorem 4.9 (Sound translation) *The library translation $\tau : \mathbb{T} \to \mathbb{C}_{\mathbb{H}}$ is sound.*

Proof. *The proof is by induction over the structure of C in $\Omega \vDash_{\mathbb{T}} \{p\}C\{q\}$ and is given in the technical report [18]. We show that translation τ has the following properties and appeal to these properties in the proof.*

Property 1 (Axiom Correctness) *For all $\Omega \in \text{PEnv}_{\mathbb{T}}$, $p, q \in \wp(\Sigma \times H_{\mathbb{T}})$, $A \in \text{Atom}_{\mathbb{T}}$,* $\Omega \vDash_{\mathbb{T}} \{p\}\ A\ \{q\} \implies \tau : \Omega \vDash_{\mathbb{T}} \{p\}\ A\ \{q\}$

This ensures that the translation correctly implements abstract atomic commands, as demonstrated intuitively throughout the paper with the `deletetree` *command.*

Property 2 (Monotonicity of** id **Relation) *For all $h_1, h_2 \in H_{\mathbb{T}}$ and $I \in \mathcal{I}_\tau$:*

$$\big\{\{h_1\}\big\}\ id\ \big\{\{h_2\}\big\} \implies \big\{\langle h_1\rangle_\tau^I\big\}\ id\ \big\{\langle h_2\rangle_\tau^I\big\}$$

A lifting of this property to abstract tree states is used in proof of rule of consequence.

Property 3 ($\ast$-Preservation) *For all $h_1, h_2 \in H_{\mathbb{T}}$ and $I \in \mathcal{I}_\tau$*

$$\langle h_1 \bullet_{\mathbb{T}} h_2\rangle_\tau^I \equiv \exists I_1, I_2.\, I_1 \cup I_2 = I \ \wedge\ \langle h_1\rangle_\tau^{I_1} \bullet_{\mathbb{C}_{\mathbb{H}}} \langle h_2\rangle_\tau^{I_2}$$

A lifting of this property to tree states is used in proof of disjoint concurrency rule.

Concluding Remarks. We have highlighted a gap in reasoning between abstract libraries and concrete implementations, due to the mismatch between the abstract connectivity of abstract data fragments and the concrete connectivity of their concrete representations. We have illustrated this gap using SSL reasoning applied to an abstract concurrent tree library $\mathbb{T}$ and CAP reasoning about a concrete implementation. This gap is closed by a refinement translation $\tau : \mathbb{T} \to \mathbb{C}_{\mathbb{H}}$, which depends crucially on an interface function I_τ mapping abstract addresses to pointer interfaces. Our SSL reasoning and results generalise to arbitrary structured data libraries, as we report in [18]. Our work concentrates on the abstract connectivity associated with our SSL reasoning. However, the gap in reasoning occurs whenever there is a difference between abstract and concrete connectivity.

References

[1] Dinsdale-Young, T., M. Dodds, P. Gardner, M. Parkinson and V. Vafeiadis, "Concurrent Abstract Predicates", ECOOP (2010), 504–528

[2] Wright, A. D., "Structural Separation Logic", PhD thesis, Imperial College London, 2013.

[3] Wheelhouse, M. J., "Segment Logic", PhD thesis, Imperial College London, 2011.

[4] Smith, G. D., "Local Reasoning about Web Programs", PhD thesis, Imperial College London, 2011.

[5] Dinsdale-Young, T., "Abstract Data and Local Reasoning", PhD thesis, Imperial College London, 2010

[6] Dinsdale-Young, T., L. Birkedal, P. Gardner, M. Parkinson and H. Yang, "Views: Compositional Reasoning for Concurrent Programs", POPL (2013), 287–300

[7] Calcagno, C., P. Gardner and U. Zarfaty, "Context logic and Tree Update", POPL (2005), 271–282

[8] Filipovic, I., P. O'Hearn, N. Torp-Smith and H. Yang, *Blaming the Client: on Data Refinement in the Presence of Pointers*, Formal Aspects of Computing **22** (2010), 547–583.

[9] Dinsdale-Young, T., P. Gardner, M. Wheelhouse, "Abstraction and Refinement for Local Reasoning", VSTTE (2010), 199–215.

[10] Calcagno, C., P. O'Hearn and H. Yang, "Local Action and Abstract Separation Logic", LICS (2007), 366–378.

[11] Bornat, R., C. Calcagno, P. O'Hearn and M. Parkinson, "Permission Accounting in Separation Logic", POPL (2005), 259–270.

[12] Gardner, P., G. Smith, M. Wheelhouse and U. Zarfaty, "Local Hoare Reasoning about DOM", PODS (2008), 261–270.

[13] Jensen, J. and L. Birkedal, “Fictional Separation Logic”, ESOP (2012), 377–396.

[14] Bornat, R., C. Calcagno and H. Yang, *Variables As Resource in Separation Logic*, Electronic Notes in Theoretical Computer Science **155** (2006), 247–276.

[15] Turon, A., D. Dreyer and L. Birkedal, *Unifying Refinement and Hoare-Style Reasoning in a Logic for Higher-Order Concurrency*.

[16] Svendsen, K., L. Birkedal and M. Parkinson, *Impredicative Concurrent Abstract Predicates*, ESOP2014.

[17] Svendsen, K, L. Birkedal and M. Parkinson, “Higher-order Concurrent Abstract Predicates”, Technical report, IT University of Copenhagen (2012).

[18] Raad, A., P. Gardner, A. Wright, M. Wheelhouse, “Abstract Local Reasoning for Concurrent Libraries (Technical Report)”, http://www.doc.ic.ac.uk/~azalea/MFPS2014/TechnicalReport.pdf.

[19] Gardner P., G. Ntzik and A. Wrigth, *Local Reasoning for POSIX File System*, ESOP (2014), 169–188.

[20] P. O’Hearn, *Resources, Concurrency and Local Reasoning*, Theor. Comput. Sci., **375** (2007), 271–307.