Under consideration for publication in Math. Struct. in Comp. Science

Abstraction and Refinement for Local Reasoning

Thomas Dinsdale-Young, Philippa Gardner and Mark Wheelhouse

Department of Computing, Imperial College London, 180 Queen's Gate, London, SW7 2AZ

Received February 2011

Local reasoning has become a well-established technique in program verification, which has been shown to be useful at many different levels of abstraction. In separation logic, we use a low-level abstraction that is close to how the machine sees the program state. In context logic, we work with high-level abstractions that are close to how the clients of modules see the program state. We apply program refinement to local reasoning, demonstrating that high-level, abstract local reasoning is sound for module implementations. We consider two approaches: one that preserves the high-level locality at the low level; and one that breaks the high-level 'fiction' of locality.

Contents

1. Introduction

Hoare logic (**?**) is an important tool for formally proving correctness properties of programs. It takes advantage of modularity by treating program fragments in terms of provable specifications. However, heap programs tend to break this type of modular reasoning by permitting pointer aliasing. For instance, the specification that a program reverses one list does not imply that it leaves a second list alone. To achieve this disjointness property, it is necessary to establish disjointness conditions throughout the proof.

O'Hearn, Reynolds, and Yang (2001) introduced separation logic for reasoning *locally* about heap programs, in order to address this problem. The fundamental principle of local reasoning is that, if we know how a local computation behaves on some state, then we can infer the behaviour when the state is extended: it simply leaves the additional state unchanged. A program is specified in terms of its *footprint* — the resource necessary for it to operate — and a *frame rule* is used to infer that any additional resource is indeed unchanged. For example, given a proof that a program reverses a list, the frame rule can directly establish that the program leaves a second disjoint list alone. Consequently, separation logic enables modular reasoning about heap programs. Calcagno, O'Hearn, and Yang (2007b) developed *abstract* separation logic to provide a general theory and semantic basis for separation logics based on variants of the heap model.

Abstraction (see *e.g.* **??**) and refinement (see *e.g.* Hoare, 1972; de Roever and Engelhardt, 1999) are also essential for modular reasoning. Abstraction takes a concrete program and produces an abstract specification; refinement takes an abstract specification and produces a correct implementation. Both approaches result in a program that correctly implements an abstract specification. Such a result is essential for modularity because it means that a program can be replaced by any other program that meets the same specification. Abstraction and refinement are well-established techniques in program verification, but have so far not been fully understood in the context of local reasoning.

Parkinson and Bierman (2005) introduced abstract predicates in separation logic to provide abstract reasoning. An abstract predicate is, to the client, an opaque object that encapsulates the unknown representation of an abstract datatype. They inherit some of the benefits of locality from separation logic: an operation on one abstract predicate leaves others alone. However, the client cannot take advantage of local behaviour that is provided by the abstraction itself.

Consider a set module. The operation of removing, say, the value 3 from the set is local at the abstract level; it is independent of whether any other value is in the set.

Yet, consider an implementation of the set as a sorted, singly-linked list in the heap, starting from address h. The operation of removing 3 from the set must traverse the list from h. The footprint, therefore, comprises the entire list segment from h up to the node with value 3. With abstract predicates, the abstract footprint corresponds to the concrete footprint and hence, in this case, includes all the elements of the set less than or equal to 3. Consequently, abstract predicates cannot be used to present a local abstract specification for removing 3.

Calcagno, Gardner, and Zarfaty (2005) introduced context logic, a generalisation of separation logic, to provide *abstract local reasoning* about abstract data structures. Context logic has been used to reason about programs that manipulate data structures, such as sequences, multisets and trees (Calcagno et al., 2007a). In particular, it has been successfully applied to reason about the W3C DOM tree update library (Gardner et al., 2008). Thus far, context logic reasoning has always been justified with respect to an operational semantics defined at the same level of abstraction as the reasoning. In this paper, we combine abstract local reasoning with data refinement, to refine abstract module specifications into correct implementations.

? (**?**; Mijajlović et al. 2004) have previously considered data refinement for local reasoning, in the context of heap programs. They observed that a client can violate a module's abstraction boundary by dereferencing pointers to the module's internal state, and thereby break the refinement between abstract and concrete module implementations. In their motivating example, a simple memory allocator, a client can violate the concrete allocator's free list through pointers to memory that has been deallocated; the abstract allocator, which maintains a free set, is unaffected by such an access, hence the refinement breaks. Their solution was to "blame the client" by introducing a modified operational semantics that treats such access violations as faulting executions. Using special simulation relations, they were able to recover soundness of data refinement. They illustrate their approach with the example of a toy memory manager. Their techniques adapt to different data store models, however, both module and client use the same model.

In this work, we apply data refinement to local reasoning, demonstrating that abstract local reasoning is sound for module implementations. By contrast with **?**, we work with the axiomatic semantics, rather than operational semantics, of the language, defining proof transformations that establish that concrete implementations simulate abstract specifications. This avoids having to consider badly behaved clients, since the proof system only makes guarantees about well behaved clients. Furthermore, the abstract and concrete levels in our refinements typically have *different* data store models, meaning that the concept of locality itself is different at each level.

The motivating example of this paper is the stepwise refinement of a tree module $\mathbb{T}$, illustrated in Fig. 1. We present two different refinements from the tree module $\mathbb{T}$ to the familiar heap module of separation logic $\mathbb{H}$. The first, τ_1, described in §5.4, uses a direct implementation of trees in the heap in which each tree node is represented by a contiguous block of heap cells.

The second refinement uses an abstract list module $\mathbb{L}$ as an intermediate step in the refinement. We first show how the tree module $\mathbb{T}$ may be correctly implemented in the combination of the heap and list modules, $\mathbb{H} + \mathbb{L}$. This step, translation τ_2, is described

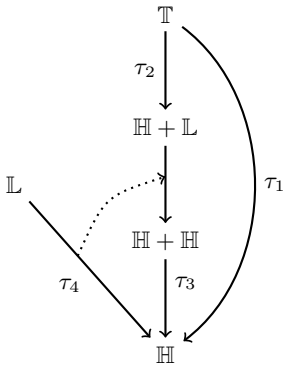

Fig. 1. Module translations presented in this paper

in §5.5. We also show that the list module $\mathbb{L}$ can be correctly implemented using the heap module $\mathbb{H}$. This is translation τ_4, which is described in §6.3. Because our approach is modular, this translation can be lifted to a translation from the combined heap and list module $\mathbb{H} + \mathbb{L}$ to the combination of two heap modules $\mathbb{H} + \mathbb{H}$. (This is illustrated by the dotted arrow in Fig. 1.) To complete the refinement, in §**??** we show that the double-heap module $\mathbb{H} + \mathbb{H}$ can be trivially implemented by the heap module $\mathbb{H}$.

Our development introduces two general techniques for verifying module implementations with respect to their local specifications, using the data refinement technique known as *forward simulation* (*L-simulation* in de Roever and Engelhardt, 1999). We introduce *locality-preserving* and *locality-breaking* translations. Locality-preserving translations, broadly speaking, relate locality at the abstract level with locality of the implementation. However, implementations typically operate on a larger state than the abstract footprint, for instance, by performing pointer surgery on the surrounding state. We introduce the notion of *crust* to capture this additional state. This crust intrudes on the context, and so breaks the disjointness that exists at the abstract level. We therefore relate abstract locality with implementation-level locality through a *fiction of disjointness*.

With locality-breaking translations, locality at the abstract level does not correspond to locality of the implementation. Even in this case, we can think about a locality-preserving translation using possibly the whole data structure as the crust. Instead, we prove soundness by establishing that the specifications of the module commands are preserved under translation in any abstract context, showing the soundness of the abstract frame rule. We thus establish a *fiction of locality* at the abstract level.

1.1. *Dedication*

This paper is dedicated to the memory of Robin Milner. Milner was an early pioneer of program verification, creating the LCF system (**?**) for assisting a user in proving properties about programs. Milner also achieved substantial research on modularity and

abstraction: he emphasised the use of program modules to provide abstract data types using ML modules (**?**); he developed full abstraction (**?**), providing a strong link between denotational and operational semantics; he introduced the theory of bisimulation for concurrent processes, for example demonstrating that a process corresponding to a specification has the same behaviour as a process describing an implementation. More recently, he was one of the leaders of the UK Grand Challenge *Global Ubiquitous Computing: Design and Science*, in which he highlighted a key aim: 'to develop a coherent informatic science whose concepts, calculi, theories and automated tools allow descriptive and predictive analysis of global ubiquitous computing at each level of abstraction.' Our work resonates well with this aim, by introducing a coherent theory of abstract local reasoning which refines abstract local specifications about modules into correct implementations.

1.2. *Acknowledgements*

We acknowledge the support of EPSRC Programme Grant "Resource Reasoning". Gardner acknowledges the support of a Microsoft/RAEng Senior Research Fellowship. Dinsdale-Young and Wheelhouse acknowledge the support of EPSRC DTA awards. We thank Mohammad Raza and Uri Zarfaty for detailed discussions of this work. In particular, some of the technical details in our locality-preserving translations come from an unpublished technical report *Reasoning about High-level Tree Update and its Low-level Implementation* written by Gardner and Zarfaty in 2008.

2. Preliminaries

We introduce the syntax and axiomatic semantics for a basic imperative programming language, which includes mutable variables and standard control-flow constructs such as while loops and procedure calls. As well as variables, programs operate on a mutable data store. The programming language is parametrised by the operations on the data store, so that it can be tailored to different domains. For example, standard commands for manipulating heaps, lists and trees.

2.1. *Programming Language*

We define the syntax of our programming language, which is parametrised by a set of basic commands Cmd, ranged over by φ. The choice of these basic commands depends on the domain over which the language is to be used: for instance, to work with the heap, commands for allocation, mutation, lookup and disposal of heap cells would be necessary; to work with a list, commands for lookup, insertion and removal of elements would be necessary; and to work with a tree, commands for lookup, node insertion and subtree deletion would be necessary.

We assume a syntax for *value expressions* $E \in \mathsf{Expr}$ that includes variable look-up, constant values and basic arithmetic operations. Similarly, we assume a syntax for *boolean*

expressions $B \in \mathsf{BExp}$ that includes equality and inequality on value expressions, and standard boolean operators.

Definition 2.1 (Programming Language Syntax). Given a set of basic commands Cmd, ranged over by φ, the language $\mathcal{L}_{\mathsf{Cmd}}$ is the set of syntactically valid programs, making use of these basic commands, ranged over by $\mathbb{C}, \mathbb{C}_1, \ldots$, defined as:

$$\begin{aligned}\mathbb{C} ::= {} & \varphi \mid \texttt{skip} \mid x := E \mid \mathbb{C}_1; \mathbb{C}_2 \\ & \mid \texttt{if } B \texttt{ then } \mathbb{C}_1 \texttt{ else } \mathbb{C}_2 \mid \texttt{while } B \texttt{ do } \mathbb{C} \\ & \mid \texttt{procs } \overrightarrow{r_1} := \mathtt{f}_1(\overrightarrow{x_1}) \ \{\mathbb{C}_1\}, \ldots, \overrightarrow{r_k} := \mathtt{f}_k(\overrightarrow{x_k}) \ \{\mathbb{C}_k\} \texttt{ in } \mathbb{C} \\ & \mid \texttt{call } r_1, \ldots, r_i := \mathtt{f}(E_1, \ldots, E_j) \mid \texttt{local } x \texttt{ in } \mathbb{C}\end{aligned}$$

where $x, r_1, \ldots \in \mathsf{Var}$ range over program variables, $\overrightarrow{x_i}, \overrightarrow{r_i} \in \mathsf{Var}^*$ range over vectors of program variables, $E, E_1, \ldots \in \mathsf{Expr}$ range over expressions, $B \in \mathsf{BExp}$ ranges over Boolean expressions, and $\mathtt{f}, \mathtt{f}_1, \ldots \in \mathsf{PName}$ range over procedure names. The names $\mathtt{f}_1, \ldots, \mathtt{f}_k$ of procedures defined in a single $\texttt{procs} - \texttt{in}$ block are required to be pairwise distinct. The parameter and return variables are required to be pairwise distinct within each procedure definition.

2.2. *State Model*

We work with multiple data structures at multiple levels of abstraction. To handle these structures in a uniform way, we model our program states using context algebras. Context algebras are a generalisation of separation algebras (Calcagno et al., 2007b) to more complex data structures. Separation algebras are based on a commutative combination of resource, whereas context algebras are instead based on non-commutative resource combinations, which is necessary to handle structured data, such as lists and trees. We will see that many interesting state models fit the pattern of a context algebra.

Definition 2.2 (Context Algebra). A *context algebra* $\mathcal{A} = (\mathsf{C}, \mathsf{D}, \circ, \bullet, \mathbf{I}, \mathbf{0})$ consists of

— a non-empty set of contexts, C,
— a non-empty set of abstract data structures, D,
— a context composition operator, $\circ : \mathsf{C} \times \mathsf{C} \rightharpoonup \mathsf{C}$,
— an application operator, $\bullet : \mathsf{C} \times \mathsf{D} \rightharpoonup \mathsf{D}$,
— a distinguished set of contexts, $\mathbf{I} \subseteq \mathsf{C}$, and
— a distinguished set of abstract data structures, $\mathbf{0} \subseteq \mathsf{D}$,

for which the following properties hold: for all $c, c', c'' \in \mathsf{C}$, $d \in \mathsf{D}$, and $i' \in \mathbf{I}$

— $c \circ (c' \circ c'') = (c \circ c') \circ c''$ (that is, composition is associative);
— $c \circ (c' \bullet d) = (c \circ c') \bullet d$ (that is, composition associates with application);
— $i \circ c$ is defined for some $i \in \mathbf{I}$, and whenever $i' \circ c$ is defined, $i' \circ c = c$ (that is, $\mathbf{I}$ is a left identity of $\circ$);
— $c \circ i$ is defined for some $i \in \mathbf{I}$, and whenever $c \circ i'$ is defined, $c \circ i = c$ (that is, $\mathbf{I}$ is a right identity of $\circ$);
— $i \bullet d$ is defined for some $i \in \mathbf{I}$, and whenever $i' \bullet d$ is defined, $i' \bullet d = d$ (that is, $\mathbf{I}$ is a left identity of $\bullet$); and

— the relation $\{(c,d) \mid \exists o \in \mathbf{0}.\ c \bullet o = d\}$ is a total surjective function.

(Undefined terms are considered equal.)

Example 1. The following are examples of context algebras:

(a) *Heaps* $h \in \mathsf{Heap}$ are defined as:

$$h ::= \mathrm{emp} \mid a \mapsto v \mid h * h$$

where $a \in \mathsf{Addr}$ ranges over unique *heap addresses*, $v \in \mathsf{Val}$ ranges over *values*, and $*$ is associative and commutative with identity emp. (Heaps are thus finite partial functions from addresses to values.) Heaps form the *heap context algebra*, $\mathcal{A}_{\mathbb{H}} = (\mathsf{Heap}, \mathsf{Heap}, *, *, \{\mathrm{emp}\}, \{\mathrm{emp}\})$. All separation algebras (Calcagno et al., 2007b) can be viewed as context algebras in this way.

(b) *Variable scopes* $\rho \in \mathsf{Scope}$ are defined as:

$$\rho ::= \mathrm{emp} \mid \boldsymbol{x} \Rrightarrow v \mid \rho * \rho$$

where $\boldsymbol{x} \in \mathsf{Var}$ ranges over unique *program variables*, $v \in \mathsf{Val}$ ranges over values, and $*$ is associative and commutative with identity emp. Variable scopes form the *variable scope context algebra*, $\mathcal{A}_{\mathsf{Scope}} = (\mathsf{Scope}, \mathsf{Scope}, *, *, \{\mathrm{emp}\}, \{\mathrm{emp}\})$. The variable scope context algebra allows us to treat variables as resource (**?**).

(c) *Lists* $l \in \mathsf{Lst}$ and *list contexts* $lc \in \mathsf{C}_{\mathsf{Lst}}$ are defined as:

$$\begin{array}{rcl} l & ::= & \varnothing \mid v \mid l_1 \cdot l_2 \\ lc & ::= & \_ \mid lc \cdot l \mid l \cdot lc \end{array}$$

where values $v \in \mathsf{Val}$ are taken to occur uniquely in each list or list context and $\cdot$ is taken to be associative with identity $\varnothing$. Context application $\bullet : \mathsf{C}_{\mathsf{Lst}} \times \mathsf{Lst} \rightharpoonup \mathsf{Lst}$ and composition $\circ : \mathsf{C}_{\mathsf{Lst}} \times \mathsf{C}_{\mathsf{Lst}} \rightharpoonup \mathsf{C}_{\mathsf{Lst}}$ are defined in terms of substitution as:

$$\begin{array}{rcl} lc \bullet l & \stackrel{\text{def}}{=} & lc[l/\_] \text{ provided that } lc[l/\_] \in \mathsf{Lst} \\ lc \circ lc' & \stackrel{\text{def}}{=} & lc[lc'/\_] \text{ provided that } lc[lc'/\_] \in \mathsf{C}_{\mathsf{Lst}}. \end{array}$$

Both operators are clearly non-commutative. Lists and list contexts form the *list context algebra*, $\mathcal{A}_{\mathsf{Lst}} = (\mathsf{C}_{\mathsf{Lst}}, \mathsf{Lst}, \circ, \bullet, \{\_\}, \{\varnothing\})$.

(d) *Trees* $t \in \mathsf{Tree}$ and *tree contexts* $c \in \mathsf{C}$ are defined as:

$$\begin{array}{rcl} t & ::= & \varnothing \mid \mathbf{a}[t] \mid t \otimes t \\ c & ::= & \_ \mid \mathbf{a}[c] \mid t \otimes c \mid c \otimes t \end{array}$$

where $\mathbf{a} \in \Sigma$ ranges over unique *node identifiers*, and $\otimes$ is associative with identity $\varnothing$. Context application $\bullet : \mathsf{C}_{\mathsf{Tree}} \times \mathsf{Tree} \rightharpoonup \mathsf{Tree}$ and composition $\circ : \mathsf{C}_{\mathsf{Tree}} \times \mathsf{C}_{\mathsf{Tree}} \rightharpoonup \mathsf{C}_{\mathsf{Tree}}$ are defined in terms of substitution as:

$$\begin{array}{rcl} c \bullet t & \stackrel{\text{def}}{=} & c[t/\_] \text{ provided that } c[t/\_] \in \mathsf{Tree} \\ c \circ c' & \stackrel{\text{def}}{=} & c[c'/\_] \text{ provided that } c[c'/\_] \in \mathsf{C}_{\mathsf{Tree}}. \end{array}$$

Both operators are clearly non-commutative. Trees and tree contexts form the *tree context algebra*, $\mathcal{A}_{\mathbb{T}} = (\mathsf{C}, \mathsf{Tree}, \circ, \bullet, \{\_\}, \{\varnothing\})$.

(e) Let $\mathcal{A}_1 = (\mathsf{C}_1, \mathsf{D}_1, \circ_1, \bullet_1, \mathbf{I}_1, \mathbf{0}_1)$ and $\mathcal{A}_2 = (\mathsf{C}_2, \mathsf{D}_2, \circ_2, \bullet_2, \mathbf{I}_2, \mathbf{0})$ be context algebras. Then their *direct product* $\mathcal{A}_1 \times \mathcal{A}_2 = (\mathsf{C}_1 \times \mathsf{C}_2, \mathsf{D}_1 \times \mathsf{D}_2, \circ_1 \times \circ_2, \bullet_1 \times \bullet_2, \mathbf{I}_1 \times \mathbf{I}_2, \mathbf{0}_1 \times \mathbf{0}_2)$ is also a context algebra[†]. For example, $\mathcal{A}_{\mathsf{Scope}} \times \mathcal{A}_{\mathbb{H}}$ and $\mathcal{A}_{\mathsf{Scope}} \times \mathcal{A}_{\mathbb{T}}$ combine variable scopes with heaps and trees, respectively.

2.3. *Axiomatic Semantics*

We define an axiomatic semantics for $\mathcal{L}_{\mathsf{Cmd}}$ as a program logic based on local Hoare reasoning. This semantics treats the space of program states as a context algebra $\mathcal{A}_{\mathsf{State}} = (\mathsf{C}_{\mathsf{State}}, \mathsf{State}, \circ, \bullet, \mathbf{I}, \mathbf{0})$, which is the product of the variable scope context algebra $\mathcal{A}_{\mathsf{Scope}}$, from Example 1(b), with some data store context algebra $\mathcal{A}_{\mathsf{Store}}$; *i.e.* $\mathcal{A}_{\mathsf{State}} = \mathcal{A}_{\mathsf{Scope}} \times \mathcal{A}_{\mathsf{Store}}$. The semantics is parametrised by the choice of $\mathcal{A}_{\mathsf{Store}}$ and the axioms given for basic commands. This gives us a fixed way of treating program variables, but allows for a flexible choice of the structure in the data store. The treatment of variables as resource (**?**) also allows us to avoid side conditions in our proof rules.

The Hoare logic judgements of our proof system make assertions about program state and have the form $\Gamma \vdash \{P\}\ \mathbb{C}\ \{Q\}$, where $P, Q \in \mathsf{State}$ are predicates, $\mathbb{C} \in \mathcal{L}_{\mathsf{Cmd}}$ is a program, and Γ is a procedure specification environment. A *procedure specification environment* associates procedure names with pairs of pre- and postconditions (parameterised by the arguments and return values of the procedure respectively). The interpretation of judegments is that, in the presence of procedures satisfying Γ, when executed from a state satisfying P, the program $\mathbb{C}$ will either diverge or terminate in a state satisfying Q.

For simplicity, we use semantic predicates as assertions (so assertions describe sets of states), rather than using logical formulae; this reflects the practice of Calcagno et al. (2007b).

Definition 2.3 (Predicates). The set of *state predicates* $\mathcal{P}(\mathsf{State})$, ranged over by $P, Q, R, P', P_1, \ldots$, is the set of sets of states. The set of *state-context predicates* $\mathcal{P}(\mathsf{C}_{\mathsf{State}})$, ranged over by $K, K', K_1, \ldots$, is the set of sets of state contexts.

To express predicates, we use standard logical notation for conjunction, disjunction, negation and quantification, which are interpreted with the usual semantics (intersection, union, *etc.*). We also lift operations on states and contexts to predicates: for instance, $a \mapsto v$ denotes the predicate $\{a \mapsto v\}$; $\exists v.\, a \mapsto v$ denotes $\{a \mapsto v \mid v \in \mathsf{Val}\}$; $P * Q$ denotes $\{d_1 * d_2 \mid d_1 \in P \text{ and } d_2 \in Q\}$; the separating application $K \bullet P$ denotes $\{c \bullet d \mid c \in K \text{ and } d \in P\}$; and so on. We use $a \mapsto -$ as shorthand for $\exists v.\, a \mapsto v$ and similarly for $\boldsymbol{x} \Rightarrow -$. We use $\prod^*$ to denote iterated $*$, and $K \multimap K'$ to denote $\{c_1 \mid \text{for all } c_2 \in K \text{ and } c \in c_1 \circ c_2,\ c \in K'\}$. We use $\equiv$ to indicate equality between predicates, $\subseteq$ to indicate that one predicate is contained with in another (the first entails the second), and $\in$ to indicate that a state or context belongs to a predicate.

Recall from our definition of $\mathcal{L}_{\mathsf{Cmd}}$ 2.1 that procedures have the form $\overrightarrow{r} := \mathtt{f}(\overrightarrow{x})$.

[†] The product of partial functions is defined pointwise in the natural fashion.

Definition 2.4 (Procedure Specification Environment). A *procedure specification* $\mathtt{f} : \mathsf{P} \rightarrowtail \mathsf{Q}$ comprises

— a procedure name $\mathtt{f} \in \mathsf{PName}$,
— a parametrised precondition $\mathsf{P} : \mathsf{Val}^n \to \mathcal{P}(\mathsf{State})$, and
— a parametrised postcondition $\mathsf{Q} : \mathsf{Val}^m \to \mathcal{P}(\mathsf{State})$,

where $\overrightarrow{r} := \mathtt{f}(\overrightarrow{x})$, $|\overrightarrow{r}| = n$ and $|\overrightarrow{x}| = m$.

A *procedure specification environment* is a set of procedure specifications. The metavariables $\Gamma, \Gamma', \ldots$ range over procedure specification environments.

Notation.
In proof judgements, Γ, Γ' stands for the union $\Gamma \cup \Gamma'$.

We assume a denotational semantics for value expressions,

$$\mathcal{E}\,[\![(\cdot)]\!] : \mathsf{Expr} \to (\mathsf{Scope} \rightharpoonup \mathsf{Val}),$$

and for boolean expressions,

$$\mathcal{B}\,[\![(\cdot)]\!] : \mathsf{BExp} \to (\mathsf{Scope} \rightharpoonup \mathsf{Bool}).$$

where $\mathsf{Bool} \stackrel{\text{def}}{=} \{\mathbf{T}, \mathbf{F}\}$ is the standard two-valued set; $\mathbf{T}$ represents that a boolean expression holds for a given variable scope, and $\mathbf{F}$ represents that it does not. Note that the semantics of an expression can be undefined on a given variable scope, for instance if some variable in the expression is not assigned in the scope, or if evaluating the expression requires division by zero.

To simplify the presentation of our inference rules we define a predicate-valued semantics for boolean expressions. This semantics interprets a boolean expression as the set of states in which that boolean expression holds.

Definition 2.5 (Predicate-Valued Semantics of Boolean Expressions). The *predicate-valued semantics of Boolean expressions*, $\mathcal{P}\,[\![(\cdot)]\!] : \mathsf{BExp} \to \mathcal{P}(\mathsf{State})$, is defined in terms of their truth-valued semantics as follows:

$$\mathcal{P}\,[\![B]\!] = \{(\rho, \chi) \mid \mathcal{B}\,[\![B]\!]\,\rho = \mathbf{T}\}\,.$$

Since the semantics of expressions is partial, it is also convenient to define safety predicates, which simply assert that the state permits the evaluation of a given expression.

Definition 2.6 (Safety Predicates). Given a value expression, $E \in \mathsf{Expr}$, the *expression safety predicate for* E, $\mathit{vsafe}(E)$, is defined as:

$$\mathit{vsafe}(E) = \{(\rho, \chi) \mid \mathcal{E}\,[\![E]\!]\,\rho \text{ is defined}\}\,.$$

Similarly, given a Boolean expression, $B \in \mathsf{BExp}$, the *expression safety predicate for* B, $\mathit{bsafe}(B)$, is defined as:

$$\mathit{bsafe}(B) = \{(\rho, \chi) \mid \mathcal{B}\,[\![B]\!]\,\rho \text{ is defined}\}\,.$$

Finally, in order to define the axiomatic semantics, we need axioms for the basic commands of the language.

Assumption 1 (Axioms for Basic Commands). Assume a set of axioms for the basic commands,

$$\textsc{Ax}\,[\![(\cdot)]\!] : \mathsf{Cmd} \to \mathcal{P}(\mathcal{P}(\mathsf{State}) \times \mathcal{P}(\mathsf{State})).$$

Definition 2.7 (Inference Rules). The Hoare logic rules for $\mathcal{L}_{\mathsf{Cmd}}$ are given in Fig. 2.

The Axiom rule allows us to use the given specifications of our basic commands and the Frame rule is the natural generalisation of the frame rule for separation algebras to context algebras. The Cons, Disj, Skip, Seq, If and While rules are standard. The Assgn, Local, PDef, PCall and PWk rules are standard, but adapted to our treatment of variables as resource. The Assgn rule not only requires the resource $\boldsymbol{x} \Rightarrow v$, but also the resource ρ containing the other variables used in E. For the Local rule, recall that the predicate P specifies a set of pairs consisting of resource from $\mathsf{D}_{\mathsf{Store}}$ and variable resource. The predicate $(\boldsymbol{x} \Rightarrow - \times \mathbf{I}_{\mathsf{Store}}) \bullet P$ therefore extends the variable component with variable $\boldsymbol{x}$ of indeterminate initial value. If a local variable block is used to re-declare a variable that is already in scope, the Frame rule must be used add the variable's outer scope after the Local rule is applied. For the PDef and PCall rules, the procedure `f` has parametrised predicates $\mathsf{P} = \lambda\overrightarrow{x}.P$ and $\mathsf{Q} = \lambda\overrightarrow{r}.Q$ as its pre- and postcondition, with $\mathsf{P}(\overrightarrow{v}) = P[\overrightarrow{v}/\overrightarrow{x}]$ and $\mathsf{Q}(\overrightarrow{w}) = Q[\overrightarrow{w}/\overrightarrow{r}]$; the parameters carry the call and return values of the procedure. The PWk rule simply allows for the procedure specification environment to be weakened (i.e. more procedure specifications can be added).

The conjunction rule is notably absent from Fig. 2.:

$$\frac{I \neq \emptyset \quad \text{for all } i \in I,\, \Gamma \vdash \{P_i\}\ \mathbb{C}\ \{Q_i\}}{\Gamma \vdash \left\{\bigwedge_{i\in I} P_i\right\}\ \mathbb{C}\ \left\{\bigwedge_{i\in I} Q_i\right\}}\ \textsc{Conj}.$$

We give conditions under which conjunction is admissible. To justify our ideas, we first study an example where conjunction is not admissible. Consider the command `allocEither()` that operates on a double-heap data store ($\mathcal{A}_{\mathsf{Store}} = \mathcal{A}_{\mathbb{H}} \times \mathcal{A}_{\mathbb{H}}$), allocating a single cell in either of the two heaps. We can specify this command by:

$$\textsc{Ax}\,[\![\boldsymbol{x} := \texttt{allocEither}()]\!] \stackrel{\text{def}}{=} \left\{ \begin{array}{l} ((\boldsymbol{x} \Rightarrow - \times \mathrm{emp} \times \mathrm{emp}), (\exists a.\, \boldsymbol{x} \Rightarrow a \times a \mapsto - \times \mathrm{emp})), \\ ((\boldsymbol{x} \Rightarrow - \times \mathrm{emp} \times \mathrm{emp}), (\exists a.\, \boldsymbol{x} \Rightarrow a \times \mathrm{emp} \times a \mapsto -)) \end{array} \right\}$$

Note that the choice of heap in which the cell is allocated is not at the discretion of the implementation, but rather at the discretion of the prover. The command exhibits *angelic nondeterminism*: that is, it is possible to prove both that the command allocates the cell in the left heap and that the command allocates the cell in the right heap. This seems paradoxical, since the program must somehow correctly guess the prover's choice.

Remark. One way of resolving this paradox is that, from the program's perspective, the two cases are actually *the same* and the distinction is only a logical abstraction. This is exactly the case for the implementation of a double heap in a single heap that we consider in §5.6.

$$\frac{(P,Q) \in \mathrm{Ax}\,[\![\varphi]\!]}{\Gamma \vdash \{P\}\ \varphi\ \{Q\}}\ \textsc{Axiom} \qquad \frac{\Gamma \vdash \{P\}\ \mathbb{C}\ \{Q\}}{\Gamma \vdash \{K \bullet P\}\ \mathbb{C}\ \{K \bullet Q\}}\ \textsc{Frame}$$

$$\frac{P \subseteq P' \quad \Gamma \vdash \{P'\}\ \mathbb{C}\ \{Q'\} \quad Q' \subseteq Q}{\Gamma \vdash \{P\}\ \mathbb{C}\ \{Q\}}\ \textsc{Cons}$$

$$\frac{\text{for all } i \in I,\ \Gamma \vdash \{P_i\}\ \mathbb{C}\ \{Q_i\}}{\Gamma \vdash \left\{\bigvee_{i\in I} P_i\right\}\ \mathbb{C}\ \left\{\bigvee_{i\in I} Q_i\right\}}\ \textsc{Disj}$$

$$\frac{}{\Gamma \vdash \{\mathbf{0}\}\ \mathtt{skip}\ \{\mathbf{0}\}}\ \textsc{Skip}$$

$$\frac{\Gamma \vdash \{P\}\ \mathbb{C}_1\ \{R\} \quad \Gamma \vdash \{R\}\ \mathbb{C}_2\ \{Q\}}{\Gamma \vdash \{P\}\ \mathbb{C}_1; \mathbb{C}_2\ \{Q\}}\ \textsc{Seq}$$

$$\frac{P \subseteq \mathit{bsafe}(B) \quad \Gamma \vdash \{P \wedge \mathcal{P}\,[\![B]\!]\}\ \mathbb{C}_1\ \{Q\} \quad \Gamma \vdash \{P \wedge \neg\mathcal{P}\,[\![B]\!]\}\ \mathbb{C}_2\ \{Q\}}{\Gamma \vdash \{P\}\ \mathtt{if}\ B\ \mathtt{then}\ \mathbb{C}_1\ \mathtt{else}\ \mathbb{C}_2\ \{Q\}}\ \textsc{If}$$

$$\frac{P \subseteq \mathit{bsafe}(B) \quad \Gamma \vdash \{P \wedge \mathcal{P}\,[\![B]\!]\}\ \mathbb{C}\ \{P\}}{\Gamma \vdash \{P\}\ \mathtt{while}\ B\ \mathtt{do}\ \mathbb{C}\ \{P \wedge \neg\mathcal{P}\,[\![B]\!]\}}\ \textsc{While}$$

$$\frac{(\boldsymbol{x} \Rightarrow v * \rho, \chi) \in \mathit{vsafe}(E)}{\Gamma \vdash \{\{(\boldsymbol{x} \Rightarrow v * \rho, \chi)\}\}\ \boldsymbol{x} := E\ \{\{(\boldsymbol{x} \Rightarrow \mathcal{E}\,[\![E]\!]\,(\boldsymbol{x} \Rightarrow v * \rho) * \rho, \chi)\}\}}\ \textsc{Assgn}$$

$$\frac{P \wedge \mathit{vsafe}(\boldsymbol{x}) \equiv \emptyset \quad \Gamma \vdash \begin{array}{c}\{(\boldsymbol{x} \Rightarrow - \times \mathbf{I}_{\mathsf{Store}}) \bullet P\}\\ \mathbb{C}\\ \{(\boldsymbol{x} \Rightarrow - \times \mathbf{I}_{\mathsf{Store}}) \bullet Q\}\end{array}}{\Gamma \vdash \{P\}\ \mathtt{local}\ \boldsymbol{x}\ \mathtt{in}\ \mathbb{C}\ \{Q\}}\ \textsc{Local}$$

$$\frac{\begin{array}{l}\text{for all } (\mathtt{f}_i : \mathsf{P} \rightarrowtail \mathsf{Q}) \in \Gamma,\ \ \Gamma', \Gamma \vdash \begin{array}{c}\{\exists \overrightarrow{v}.\, \overrightarrow{\boldsymbol{x}_i} \Rightarrow \overrightarrow{v} * \overrightarrow{\boldsymbol{r}_i} \Rightarrow - \times \mathsf{P}(\overrightarrow{v})\}\\ \mathbb{C}_i\\ \{\exists \overrightarrow{w}.\, \overrightarrow{\boldsymbol{x}_i} \Rightarrow - * \overrightarrow{\boldsymbol{r}_i} \Rightarrow \overrightarrow{w} \times \mathsf{Q}(\overrightarrow{w})\}\end{array}\\ \quad \text{for all } \mathtt{f} : \mathsf{P} \rightarrowtail \mathsf{Q} \in \Gamma, \text{ there exists } i \text{ s.t. } \mathtt{f} = \mathtt{f}_i\\ \quad\quad \text{for all } \mathtt{f} : \mathsf{P} \rightarrowtail \mathsf{Q} \in \Gamma', \text{ for all } i,\ \mathtt{f} \neq \mathtt{f}_i\\ \quad\quad\quad \Gamma', \Gamma \vdash \{P\}\ \mathbb{C}\ \{Q\}\end{array}}{\Gamma' \vdash \{P\}\ \mathtt{procs}\ \overrightarrow{\boldsymbol{r}_1} := \mathtt{f}_1(\overrightarrow{\boldsymbol{x}_1})\ \{\mathbb{C}_1\}, \ldots, \overrightarrow{\boldsymbol{r}_k} := \mathtt{f}_k(\overrightarrow{\boldsymbol{x}_k})\ \{\mathbb{C}_k\}\ \mathtt{in}\ \mathbb{C}\ \{Q\}}\ \textsc{PDef}$$

$$\frac{\{(\overrightarrow{\boldsymbol{r}} \Rightarrow \overrightarrow{v} * \rho)\} \times \mathsf{Store} \subseteq \mathit{vsafe}(\overrightarrow{E})}{\Gamma, \mathtt{f} : \mathsf{P} \rightarrowtail \mathsf{Q} \vdash \begin{array}{c}\left\{\{(\overrightarrow{\boldsymbol{r}} \Rightarrow \overrightarrow{v} * \rho)\} \times \mathsf{P}\left(\overrightarrow{\mathcal{E}\,[\![E]\!]}(\overrightarrow{\boldsymbol{r}} \Rightarrow \overrightarrow{v} * \rho)\right)\right\}\\ \mathtt{call}\ \ \overrightarrow{\boldsymbol{r}} := \mathtt{f}(\overrightarrow{E})\\ \{\exists \overrightarrow{w}.\, \{(\overrightarrow{\boldsymbol{r}} \Rightarrow \overrightarrow{w} * \rho)\} \times \mathsf{Q}\,(\overrightarrow{w})\}\end{array}}\ \textsc{PCall}$$

$$\frac{\Gamma \vdash \{P\}\ \mathbb{C}\ \{Q\}}{\Gamma, \Gamma' \vdash \{P\}\ \mathbb{C}\ \{Q\}}\ \textsc{PWk}$$

Fig. 2. Local Hoare logic rules for $\mathcal{L}_{\mathsf{Cmd}}$

The conjunction rule is not compatible with angelic nondeterminism, as illustrated by the following derivation:

$$\dfrac{\dfrac{}{\begin{array}{c}\{\boldsymbol{x} \Rightarrow - \times \mathrm{emp} \times \mathrm{emp}\} \\ \boldsymbol{x} := \texttt{allocEither}() \\ \{\exists a.\, \boldsymbol{x} \Rightarrow a \times a \mapsto - \times \mathrm{emp}\}\end{array}}\ \textsc{Axiom} \qquad \dfrac{}{\begin{array}{c}\{\boldsymbol{x} \Rightarrow - \times \mathrm{emp} \times \mathrm{emp}\} \\ \boldsymbol{x} := \texttt{allocEither}() \\ \{\exists a.\, \boldsymbol{x} \Rightarrow a \times \mathrm{emp} \times a \mapsto -\}\end{array}}\ \textsc{Axiom}}{\{\boldsymbol{x} \Rightarrow - \times \mathrm{emp} \times \mathrm{emp}\}\ \boldsymbol{x} := \texttt{allocEither}()\ \{\mathrm{false}\}}\ \textsc{Conj}$$

With Conj, it must be the case that `allocEither` diverges. Without Conj, we cannot draw the same conclusion.

The following two conditions on basic command $\varphi \in \mathsf{Cmd}$ are sufficient to establish that the command does not exhibit angelic nondeterminism:

— for all $(P, Q), (P', Q') \in \mathrm{Ax}\,[\![\varphi]\!]$ with $(P, Q) \neq (P', Q')$, $P \wedge P' \equiv \emptyset$; and
— the predicate $\bigvee \{P \mid (P, Q) \in \mathrm{Ax}\,[\![\varphi]\!]\}$ is precise[‡].

These conditions imply that at most one axiom describes the behaviour of the command from any given state, and hence the conjunction rule cannot be used to derive a stronger postcondition for any of the basic commands. These conditions hold for all of the basic module commands considered in this paper; they are not difficult to check. None of the program constructions introduces angelic nondeterminism [§]. Conj is therefore admissible for all of the modules considered here. Since nothing is gained by the inclusion of the conjunction rule, its omission is justified. When we later discuss our reasoning techniques in §5 and §6, we briefly consider the consequences of including the conjunction rule in our theory.

3. Abstract Modules

An abstract module is a collection of operations on some abstract state model. For example, a heap module typically provides operations that allocate and dispose blocks of heap cells, and that fetch and mutate values stored in heap cells. Similarly, a list module provides operations for adding, removing and querying list elements, while a tree module provides operations for traversing the tree structure, and adding, removing and moving nodes or subtrees.

The programming language that we introduced in the previous section can be instantiated for such abstract modules by the choice of basic commands Cmd, data store context algebra $\mathcal{A}_{\mathsf{Store}} = (\mathsf{C}_{\mathsf{Store}}, \mathsf{Store}, \circ, \bullet, \mathbf{I}_{\mathsf{Store}}, \mathbf{0}_{\mathsf{Store}})$, and axiomatisation of the basic commands $\mathrm{Ax}\,[\![(\cdot)]\!]$. Together, these three parameters constitute the notion of an abstract module in our formalism.

Definition 3.1 (Abstract Module). An *abstract module*

$$\mathbb{A} = (\mathsf{Cmd}_{\mathbb{A}}, \mathcal{A}_{\mathbb{A}}, \mathrm{Ax}\,[\![(\cdot)]\!]_{\mathbb{A}})$$

[‡] Predicate P is *precise* if, for every $d \in \mathsf{State}$ there is at most one $c \in \mathsf{C}_{\mathsf{State}}$ and $d' \in P$ such that $d = c \bullet d'$.

[§] Proving this statement is rather involved, especially for recursion and looping. The proof invokes Tarski's fixed-point theorem and therefore requires a proof of monotonicity of programs.

consists of:

— a set of basic commands, $\mathsf{Cmd}_{\mathbb{A}}$;
— a context algebra $\mathcal{A}_{\mathbb{A}} = (\mathsf{C}_{\mathbb{A}}, \mathsf{D}_{\mathbb{A}}, \circ_{\mathbb{A}}, \bullet_{\mathbb{A}}, \mathbf{I}_{\mathbb{A}}, \mathbf{0}_{\mathbb{A}})$; and
— an axiomatisation for the basic commands

$$\mathrm{Ax}\,[\![(\cdot)]\!]_{\mathbb{A}} : \mathsf{Cmd}_{\mathbb{A}} \to \mathcal{P}(\mathcal{P}(\mathsf{State}_{\mathbb{A}}) \times \mathcal{P}(\mathsf{State}_{\mathbb{A}})),$$

where $\mathsf{State}_{\mathbb{A}} = \mathsf{Scope} \times \mathsf{D}_{\mathbb{A}}$.

Notation.
We denote the language determined by the abstract module $\mathbb{A}$ as $\mathcal{L}_{\mathbb{A}}$ (this is in fact $\mathcal{L}_{\mathsf{Cmd}_{\mathbb{A}}}$). We denote the axiomatic semantic judgement determined by the abstract module $\mathbb{A}$ as $\vdash_{\mathbb{A}}$. When the abstract module $\mathbb{A}$ can be inferred from context, the subscript $_{\mathbb{A}}$ may be dropped.

In this section, we consider a number of different abstract modules, including heaps (the basis of separation logic (Ishtiaq and O'Hearn, 2001; Reynolds, 2002)), trees (the original motivation for context logic (Calcagno, Gardner, and Zarfaty, 2005)) and lists. A number of other abstract modules that are easily encoded in our formalism have been considered in the literature, such as heap with free set (Raza and Gardner, 2009), a fragment of the W3C DOM (Gardner, Smith, Wheelhouse, and Zarfaty, 2008), and tree segments (Gardner and Wheelhouse, 2009).

3.1. *Heap Module*

The abstract heap module $\mathbb{H} \stackrel{\text{def}}{=} (\mathsf{Cmd}_{\mathbb{H}}, \mathcal{A}_{\mathbb{H}}, \mathrm{Ax}\,[\![(\cdot)]\!]_{\mathbb{H}})$ should be familiar to aficionados of separation logic; its commands consist of heap allocation, disposal, mutation and lookup. Heaps are modelled as finite partial functions from heap addresses (Addr) to values (Val). The address set is assumed to be the positive integers, *i.e.* $\mathsf{Addr} = \mathbb{Z}^+$, and contained within the value set, *i.e.* $\mathsf{Addr} \subseteq \mathsf{Val}$. This enables program variables and heap cells to hold pointers to other heap cells and arithmetic operations to be performed on heap addresses (pointer arithmetic).

The constant value $\boldsymbol{nil}$, the *null reference*, is used in situations where a heap reference could occur, to indicate the absence of such a reference. It is therefore necessary that $\boldsymbol{nil} \notin \mathsf{Addr}$, so that it cannot be confused with a valid heap reference, and that $\boldsymbol{nil} \in \mathsf{Val}$, so that it may be stored in variables, heap cells, *etc.*. In particular, we take $\boldsymbol{nil} = 0$. The set $\mathsf{Addr}_{\boldsymbol{nil}} \stackrel{\text{def}}{=} \mathsf{Addr} \cup \{\boldsymbol{nil}\}$ consists of all valid references and the null reference.

Definition 3.2 (Heap Update Commands). The set of *heap update commands* $\mathsf{Cmd}_{\mathbb{H}}$, ranged over by φ, is defined as follows:

$$\varphi ::= \boldsymbol{x} := \mathtt{alloc}(E) \;\mid\; \mathtt{dispose}(E_1, E_2) \;\mid\; [E_1] := E_2 \;\mid\; \boldsymbol{x} := [E]$$

where $\boldsymbol{x} \in \mathsf{Var}$ ranges over variables and $E, E_1, E_2 \in \mathsf{Expr}$ range over value expressions.

The intuitive meaning of these commands, which will be realised by their axiomatic semantics, is as follows:

— $\boldsymbol{x} := \texttt{alloc}(E)$ allocates a contiguous block of cells in the heap of length E, returning the address of the first cell in $\boldsymbol{x}$;
— $\texttt{dispose}(E_1, E_2)$ deallocates a contiguous block of cells in the heap at address E_1 of length E_2;
— $[E_1] := E_2$ stores the value E_2 in the heap cell at address E_1; and
— $\boldsymbol{x} := [E]$ loads the contents of the heap cell at address E_1 into $\boldsymbol{x}$.

The data model for heaps, which we have already seen in Example 1(a), represents heaps as, effectively, partial functions from heap addresses to values. In this model, heaps are interpreted as resources: the cells that have values specified in the heap are the resources available to the program. Loads and stores can only be performed on heap cells that are available to the program; allocation makes new heap cells available; and deallocation makes available heap cells unavailable.

Definition 3.3 (Heap Context Algebra). The *heap context algebra* is:

$$\mathcal{A}_{\mathbb{H}} \stackrel{\text{def}}{=} (\mathsf{Heap}, \mathsf{Heap}, *, *, \{\text{emp}\}, \{\text{emp}\})$$

as given in Example 1(a).

Definition 3.4 (Heap Axiomatisation).
The *heap axiomatisation*, $\text{Ax}\,[\![(\cdot)]\!]_{\mathbb{H}} : \mathsf{Cmd}_{\mathbb{H}} \to \mathcal{P}(\mathcal{P}(\mathsf{Scope} \times \mathsf{Heap}) \times \mathcal{P}(\mathsf{Scope} \times \mathsf{Heap}))$ is defined as follows:

$$\text{Ax}\,[\![\boldsymbol{x} := \texttt{alloc}(E)]\!]_{\mathbb{H}} \stackrel{\text{def}}{=} \left\{ \left(\begin{array}{l} (\boldsymbol{x} \Rightarrow v * \rho \times \text{emp} \wedge w \geq 1), \\ \left(\begin{array}{l} \exists a.\, \boldsymbol{x} \Rightarrow a * \rho \times \\ \quad a \mapsto - * \cdots * (a + w) \mapsto - \end{array} \right) \end{array} \right) \;\middle|\; w = \mathcal{E}\,[\![E]\!]\,(\boldsymbol{x} \Rightarrow v * \rho) \right\}$$

$$\text{Ax}\,[\![\texttt{dispose}(E_1, E_2)]\!]_{\mathbb{H}} \stackrel{\text{def}}{=} \left\{ \left((\rho \times a \mapsto - * \cdots * (a + v) \mapsto -), (\rho \times \text{emp}) \right) \;\middle|\; \begin{array}{r} a = \mathcal{E}\,[\![E_1]\!]\,\rho \text{ and} \\ v = \mathcal{E}\,[\![E_2]\!]\,\rho \end{array} \right\}$$

$$\text{Ax}\,[\![[E_1] := E_2]\!]_{\mathbb{H}} \stackrel{\text{def}}{=} \left\{ \left((\rho \times a \mapsto -), (\rho \times a \mapsto v) \right) \mid a = \mathcal{E}\,[\![E_1]\!]\,\rho \text{ and } v = \mathcal{E}\,[\![E_2]\!]\,\rho \right\}$$

$$\text{Ax}\,[\![\boldsymbol{x} := [E]]\!]_{\mathbb{H}} \stackrel{\text{def}}{=} \left\{ \left((\boldsymbol{x} \Rightarrow v * \rho \times a \mapsto w), (\boldsymbol{x} \Rightarrow w * \rho \times a \mapsto w) \right) \mid a = \mathcal{E}\,[\![E]\!]\,\rho \right\}$$

3.2. *Tree Module*

The abstract tree module $\mathbb{T} \stackrel{\text{def}}{=} (\mathsf{Cmd}_{\mathbb{T}}, \mathcal{A}_{\mathbb{T}}, \text{Ax}\,[\![(\cdot)]\!]_{\mathbb{T}})$ should be familiar to adherents of context logic; its commands consist of node-relative traversal, node creation and subtree deletion. The tree model consists of *uniquely-labelled* trees, where each label may only

occur once in any given tree or context. This ensures that nodes in a tree are uniquely addressable by their labels. It is therefore assumed that the set of tree labels, Σ, is contained with in the value set, Val, *i.e.* $\Sigma \subseteq \mathsf{Val}$. It is also assumed that $\mathsf{Addr} \subseteq \Sigma$, in order that heap addresses can be used to implement node addresses.

Definition 3.5 (Tree Update Commands). The set of *tree update commands* $\mathsf{Cmd}_{\mathbb{T}}$, ranged over by φ, is defined as follows:

$$\begin{array}{rl} \varphi ::= & \boldsymbol{x} := \mathtt{getUp}(E) \;\mid\; \boldsymbol{x} := \mathtt{getLeft}(E) \;\mid\; \boldsymbol{x} := \mathtt{getRight}(E) \\ \mid & \boldsymbol{x} := \mathtt{getFirst}(E) \;\mid\; \boldsymbol{x} := \mathtt{getLast}(E) \\ \mid & \mathtt{newNodeAfter}(E) \;\mid\; \mathtt{deleteTree}(E) \end{array}$$

where $\boldsymbol{x} \in \mathsf{Var}$ ranges over variables and $E \in \mathsf{Expr}$ ranges over value expressions.

The intuitive meaning of these commands is as follows:

— $\mathtt{getUp}(E)$, $\mathtt{getLeft}(E)$ and $\mathtt{getRight}(E)$ retrieve, respectively, the identifier of the immediate parent, left sibling and right sibling (if any) of the node identified by E;
— $\mathtt{getFirst}(E)$ and $\mathtt{getLast}(E)$ retrieve, respectively, the identifiers of the first and last children (if any) of the node identified by E;
— $\mathtt{newNodeAfter}(E)$ creates a new node with some fresh identifier and no children, which inserted into the tree as the right sibling of the node identified by E; and
— $\mathtt{deleteTree}(E)$ deletes the entire subtree rooted at the node identified by E.

The data model for trees, which we have already seen in Example 1(d), represents trees as uniquely-labelled abstract forests. These trees are divisible into contexts and subtrees, with subtrees always being trees in their own right. We thus give the axiomatisation of the tree update commands in terms of such complete trees, although they may actually be subtrees of bigger trees. (Gardner and Wheelhouse (2009) specify tree commands in terms of *tree segments*, which are fragments of trees that need not be trees in their own right. We work with subtrees here for simplicity of exposition.)

Definition 3.6 (Uniquely-Labelled Tree Context Algebra). The *tree context algebra* is:

$$\mathcal{A}_{\mathbb{T}} \stackrel{\text{def}}{=} (\mathsf{C}_{\mathsf{Tree}}, \mathsf{Tree}, \circ, \bullet, \{\_\}, \{\mathbf{0}\})$$

as given in Example 1(d).

Definition 3.7 (Tree Axiomatisation).
The *tree axiomatisation*, $\mathrm{Ax}\,[\![(\cdot)]\!]_{\mathbb{T}} : \mathsf{Cmd}_{\mathbb{T}} \to \mathcal{P}(\mathcal{P}(\mathsf{Scope} \times \mathsf{Tree}) \times \mathcal{P}(\mathsf{Scope} \times \mathsf{Tree}))$ is defined as follows:

$$\mathrm{Ax}\,[\![\boldsymbol{x} := \mathtt{getUp}(E)]\!]_{\mathbb{T}} \stackrel{\text{def}}{=} \left\{ \left(\begin{array}{c} (\boldsymbol{x} \Rrightarrow v * \rho \times \mathbf{a}[t_1 \otimes \mathbf{b}[t_2] \otimes t_3])\,, \\ (\boldsymbol{x} \Rrightarrow \mathbf{a} * \rho \times \mathbf{a}[t_1 \otimes \mathbf{b}[t_2] \otimes t_3]) \end{array} \right) \;\middle|\; \mathcal{E}\,[\![E]\!]\,(\boldsymbol{x} \Rrightarrow v * \rho) = \mathbf{b} \right\}$$

$$\mathrm{Ax}\,[\![\boldsymbol{x} := \texttt{getLeft}(E)]\!]_{\mathbb{T}} \stackrel{\text{def}}{=} \left\{ \left(\begin{array}{l} (\boldsymbol{x} \Rightarrow v * \rho \times \mathbf{a}[t_1] \otimes \mathbf{b}[t_2])\,, \\ (\boldsymbol{x} \Rightarrow \mathbf{a} * \rho \times \mathbf{a}[t_1] \otimes \mathbf{b}[t_2]) \end{array} \right) \;\middle|\; \mathcal{E}\,[\![E]\!]\,(\boldsymbol{x} \Rightarrow v * \rho) = \mathbf{b} \right\} \cup \left\{ \left(\begin{array}{l} (\boldsymbol{x} \Rightarrow v * \rho \times \mathbf{a}[\mathbf{b}[t_1] \otimes t_2])\,, \\ (\boldsymbol{x} \Rightarrow \boldsymbol{nil} * \rho \times \mathbf{a}[\mathbf{b}[t_1] \otimes t_2]) \end{array} \right) \;\middle|\; \mathcal{E}\,[\![E]\!]\,(\boldsymbol{x} \Rightarrow v * \rho) = \mathbf{b} \right\}$$

$$\mathrm{Ax}\,[\![\boldsymbol{x} := \texttt{getRight}(E)]\!]_{\mathbb{T}} \stackrel{\text{def}}{=} \left\{ \left(\begin{array}{l} (\boldsymbol{x} \Rightarrow v * \rho \times \mathbf{b}[t_1] \otimes \mathbf{a}[t_2])\,, \\ (\boldsymbol{x} \Rightarrow \mathbf{a} * \rho \times \mathbf{b}[t_1] \otimes \mathbf{a}[t_2]) \end{array} \right) \;\middle|\; \mathcal{E}\,[\![E]\!]\,(\boldsymbol{x} \Rightarrow v * \rho) = \mathbf{b} \right\} \cup \left\{ \left(\begin{array}{l} (\boldsymbol{x} \Rightarrow v * \rho \times \mathbf{a}[t_1 \otimes \mathbf{b}[t_2]])\,, \\ (\boldsymbol{x} \Rightarrow \boldsymbol{nil} * \rho \times \mathbf{a}[t_1 \otimes \mathbf{b}[t_2]]) \end{array} \right) \;\middle|\; \mathcal{E}\,[\![E]\!]\,(\boldsymbol{x} \Rightarrow v * \rho) = \mathbf{b} \right\}$$

$$\mathrm{Ax}\,[\![\boldsymbol{x} := \texttt{getFirst}(E)]\!]_{\mathbb{T}} \stackrel{\text{def}}{=} \left\{ \left(\begin{array}{l} (\boldsymbol{x} \Rightarrow v * \rho \times \mathbf{a}[\mathbf{b}[t_1] \otimes t_2])\,, \\ (\boldsymbol{x} \Rightarrow \mathbf{b} * \rho \times \mathbf{a}[\mathbf{b}[t_1] \otimes t_2]) \end{array} \right) \;\middle|\; \mathcal{E}\,[\![E]\!]\,(\boldsymbol{x} \Rightarrow v * \rho) = \mathbf{a} \right\} \cup \left\{ \left(\begin{array}{l} (\boldsymbol{x} \Rightarrow v * \rho \times \mathbf{a}[\mathbf{0}])\,, \\ (\boldsymbol{x} \Rightarrow \boldsymbol{nil} * \rho \times \mathbf{a}[\mathbf{0}]) \end{array} \right) \;\middle|\; \mathcal{E}\,[\![E]\!]\,(\boldsymbol{x} \Rightarrow v * \rho) = \mathbf{a} \right\}$$

$$\mathrm{Ax}\,[\![\boldsymbol{x} := \texttt{getLast}(E)]\!]_{\mathbb{T}} \stackrel{\text{def}}{=} \left\{ \left(\begin{array}{l} (\boldsymbol{x} \Rightarrow v * \rho \times \mathbf{a}[t_1 \otimes \mathbf{b}[t_2]])\,, \\ (\boldsymbol{x} \Rightarrow \mathbf{b} * \rho \times \mathbf{a}[t_1 \otimes \mathbf{b}[t_2]]) \end{array} \right) \;\middle|\; \mathcal{E}\,[\![E]\!]\,(\boldsymbol{x} \Rightarrow v * \rho) = \mathbf{a} \right\} \cup \left\{ \left(\begin{array}{l} (\boldsymbol{x} \Rightarrow v * \rho \times \mathbf{a}[\mathbf{0}])\,, \\ (\boldsymbol{x} \Rightarrow \boldsymbol{nil} * \rho \times \mathbf{a}[\mathbf{0}]) \end{array} \right) \;\middle|\; \mathcal{E}\,[\![E]\!]\,(\boldsymbol{x} \Rightarrow v * \rho) = \mathbf{a} \right\}$$

$$\mathrm{Ax}\,[\![\texttt{newNodeAfter}(E)]\!]_{\mathbb{T}} \stackrel{\text{def}}{=} \left\{ \left((\rho \times \mathbf{a}[t])\,, (\exists \mathbf{b}.\, \rho \times \mathbf{a}[t] \times \mathbf{b}[\mathbf{0}]) \right) \;\middle|\; \mathcal{E}\,[\![E]\!]\,(\boldsymbol{x} \Rightarrow v * \rho) = \mathbf{a} \right\}$$

$$\mathrm{Ax}\,[\![\texttt{deleteTree}(E)]\!]_{\mathbb{T}} \stackrel{\text{def}}{=} \left\{ \left((\rho \times \mathbf{a}[t])\,, (\rho \times \mathbf{0}) \right) \;\middle|\; \mathcal{E}\,[\![E]\!]\,(\boldsymbol{x} \Rightarrow v * \rho) = \mathbf{a} \right\}$$

3.3. *List Module*

This list module $\mathbb{L} \stackrel{\text{def}}{=} (\mathsf{Cmd}_{\mathbb{L}}, \mathcal{A}_{\mathbb{L}}, \mathrm{Ax}\,[\![(\cdot)]\!]_{\mathbb{L}})$ is an example of a somewhat more exotic abstract module. The module provides an addressable set of lists of unique elements, called a *list store*. Each list can be manipulated independently in a number of ways, new lists can be constructed and existing lists can be deleted.

Definition 3.8 (List Update Commands). The set of *list commands* $\mathsf{Cmd}_{\mathbb{L}}$, ranged over by φ, is defined as follows:

$$\begin{aligned}\varphi ::= {} & \boldsymbol{x} := E.\texttt{getHead}() \mid \boldsymbol{x} := E.\texttt{getTail}() \\ & \mid \boldsymbol{x} := E_1.\texttt{getNext}(E_2) \mid \boldsymbol{x} := E_1.\texttt{getPrev}(E_2) \\ & \mid \boldsymbol{x} := E.\texttt{pop}() \mid E_1.\texttt{push}(E_2) \\ & \mid E_1.\texttt{remove}(E_2) \mid E_1.\texttt{insert}(E_2, E_3) \\ & \mid \boldsymbol{x} := \texttt{newList}() \mid \texttt{deleteList}(E)\end{aligned}$$

where $\boldsymbol{x} \in \mathsf{Var}$ ranges over variables and $E, E_1, \ldots \in \mathsf{Expr}$ range over value expressions.

The intuitive meaning of these commands is as follows:

— $E.\texttt{getHead}()$ and $E.\texttt{getTail}()$ retrieve, respectively, the first and last elements (if any) of the list identified by E;
— $E_1.\texttt{getNext}(E_2)$ and $E_1.\texttt{getPrev}(E_2)$ retrieve, respectively, the elements (if any) following and preceding the element E_2 in the list identified by E_1 (if E_2 is not in the list, the behaviour of these commands is undefined);
— $E.\texttt{pop}()$ retrieves and removes the first element of the list identified by E (if the list is empty, the behaviour is undefined);
— $E_1.\texttt{push}(E_2)$ adds the element E_2 to the start of the list identified by E_1;
— $E_1.\texttt{remove}(E_2)$ removes the element E_2 from the list identified by E_1 (if E_2 is not in the list, the behaviour is undefined);
— $E_1.\texttt{insert}(E_2, E_3)$ inserts the element E_3 immediately following E_2 in the list identified by E_1 (if E_2 is not in the list, the behaviour is undefined);
— $\texttt{newList}()$ creates a new list, initially empty, and returns its address; and
— $\texttt{deleteList}(E)$ deletes the list identified by E.

We require that elements occur at most once in any given list. Thus `getNext`, `getPrevious` and `insert` are unambiguous and the behaviour of `push` and `insert` is undefined if they are used to insert elements that are already present in the list.

The list context algebra of Example 1(c) is not adequate as a data model for the list module, since it only models a single list. Instead, we define a *list-store context algebra*, which models multiple independent lists. List stores are similar to heaps in the sense that they are finite maps from addresses to values, except that now the values have intrinsic structure: they are lists of unique elements.

Each list in the store is a finite sequence of values, each of which occurs only once in the list. Lists may be extended by applying contexts, for instance, $v_1 \cdot - \cdot v_2 \bullet w_1 \cdot w_2 = v_1 \cdot w_1 \cdot w_2 \cdot v_2$. However, lists may also be *completed*, which means they cannot be extended by applying a context. Completed lists are indicated by square brackets, and the result of applying a context is undefined. For example, $v_1 \cdot - \cdot v_2 \bullet [w_1 \cdot w_2]$ is undefined.

It is necessary to deal with complete lists in order to specify a number of the update and lookup commands on lists. For example, `getFirst` returns the first item in a list. Given the partial list $v_1 \cdot v_2$, it is not clear that v_1 is the first element of the list. Indeed, in the context $w \cdot -$ it is certainly not the first element. However, given the completed list $[v_1 \cdot v_2]$ it is completely certain that v_1 is the first element.

Remark. It is quite possible to define list stores to allow the list to be open at one end but not the other. For example, $[v_1$ would allow additional list elements to be added by context application to the right of v_1, but not to the left. We have chosen not to take this approach in order to avoid further complicating an already-complex definition.

Recall the definition of lists and list contexts from Example 1(c):

Definition 3.9 (List and Contexts). The set of *lists* Lst, ranged over by $l, l_1, \ldots$ and the set of *list contexts* $\mathsf{C}_{\mathsf{Lst}}$, ranged over by $lc, lc_1, \ldots$, are defined inductively as follows:

$$\begin{array}{rcl} l & ::= & \varnothing \mid v \mid l_1 \cdot l_2 \\ lc & ::= & - \mid lc \cdot l \mid l \cdot lc \end{array}$$

where values $v \in \mathsf{Val}$ are taken to occur uniquely in each list or list context and $\cdot$ is taken to be associative with identity $\varnothing$.

We defined list stores and their contexts in terms lists and list contexts.

Definition 3.10 (List Stores and Contexts). The set of *list stores* LStore, ranged over by $ls, ls_1, \ldots$ and the set of *list store contexts* $\mathsf{C}_{\mathsf{LStore}}$, ranged over by $lsc, lsc_1, \ldots$, are defined inductively as follows:

$$\begin{array}{rcl} ls & ::= & \mathrm{emp} \mid a \Mapsto l \mid a \Mapsto [l] \mid ls_1 * ls_2 \\ lsc & ::= & \mathrm{emp} \mid a \Mapsto l \mid a \Mapsto [l] \mid a \Mapsto lc \mid a \Mapsto [lc] \mid lsc_1 * lsc_2 \end{array}$$

where addresses $a \in \mathsf{Addr}$ are taken to occur uniquely in each list store or list store context and $*$ is taken to be associative and commutative with identity emp.

Since list stores are structured like heaps we need a notion of context application that will allow list stores to be split in the same fashion as heaps, for instance

$$(a_1 \Mapsto v_1 \cdot v_2 \cdot v_3) * (a_2 \Mapsto w_1 \cdot v_1) \quad = \quad (a_1 \Mapsto v_1 \cdot v_2 \cdot v_3) \bullet (a_2 \Mapsto w_1 \cdot v_1).$$

We also want context application to allow splitting within lists themselves, as was possible with the list context algebra, so, for example

$$a_1 \Mapsto v_1 \cdot v_2 \cdot v_3 \quad = \quad a_1 \Mapsto v_1 \cdot - \cdot v_3 \bullet a_1 \Mapsto v_2.$$

Since such applications can be nested, the associativity requirements on context algebras mean that it must be possible to split within multiple lists at the same time, as in

$$\begin{array}{l} a_1 \Mapsto v_1 \cdot v_2 \cdot v_3 * a_2 \Mapsto w_1 \cdot v_1 \\ \qquad = \; a_1 \Mapsto v_1 \cdot - \cdot v_3 \bullet (a_1 \Mapsto v_2 * a_2 \Mapsto w_1 \cdot v_1) \\ \qquad = \; a_1 \Mapsto v_1 \cdot - \cdot v_3 \bullet (a_2 \Mapsto - \cdot v_1 \bullet (a_1 \Mapsto v_2 * a_2 \Mapsto w_1)) \\ \qquad = \; (a_1 \Mapsto v_1 \cdot - \cdot v_3 \circ a_2 \Mapsto - \cdot v_1) \bullet (a_1 \Mapsto v_2 * a_2 \Mapsto w_1) \\ \qquad = \; (a_1 \Mapsto v_1 \cdot - \cdot v_3 * a_2 \Mapsto - \cdot v_1) \bullet (a_1 \Mapsto v_2 * a_2 \Mapsto w_1). \end{array}$$

(In a sense, the context $a_1 \Mapsto v_1 \cdot - \cdot v_3 * a_2 \Mapsto - \cdot v_1$ above can be seen as a multi-holed context — after all, it certainly has multiple holes! Yet in applying it to a list store, both of the holes must be filled up at the same time. We may see this as the context having multiple 'list holes' but a single 'list store hole'.)

Completed lists cannot be extended by application, but they can be split, as in

$$a_2 \Mapsto [w_1 \cdot w_2] \quad = \quad a_2 \Mapsto [w_1 \cdot \_] \bullet a_2 \Mapsto w_2.$$

All of these properties are embodied in the definition of the application and composition operators.

Definition 3.11 (Application and Composition). The application of list store contexts to list stores $\bullet : \mathsf{C}_{\mathsf{LStore}} \times \mathsf{LStore} \rightharpoonup \mathsf{LStore}$ is defined inductively by:

$$\begin{array}{rcl}
\mathrm{emp} \bullet ls & \stackrel{\text{def}}{=} & ls \\
(lsc * a \Mapsto l) \bullet ls & \stackrel{\text{def}}{=} & (lsc \bullet ls) * a \Mapsto l \\
(lsc * a \Mapsto [l]) \bullet ls & \stackrel{\text{def}}{=} & (lsc \bullet ls) * a \Mapsto [l] \\
(lsc * a \Mapsto lc) \bullet (ls * a \Mapsto l) & \stackrel{\text{def}}{=} & (lsc \bullet ls) * a \Mapsto lc[l/_] \\
(lsc * a \Mapsto [lc]) \bullet (ls * a \Mapsto l) & \stackrel{\text{def}}{=} & (lsc \bullet ls) * a \Mapsto [lc[l/_]]
\end{array}$$

where $lc[l/\_]$ denotes the substitution of l for the context hole in lc. The result of the application is undefined when either the right-hand side is badly formed or no case applies.

The composition of list store contexts $\circ : \mathsf{C}_{\mathsf{LStore}} \times \mathsf{C}_{\mathsf{LStore}} \rightharpoonup \mathsf{C}_{\mathsf{LStore}}$ is defined inductively by:

$$\begin{array}{rcl}
\mathrm{emp} \circ lsc' & \stackrel{\text{def}}{=} & lsc' \\
(lsc * a \Mapsto l) \circ lsc' & \stackrel{\text{def}}{=} & (lsc \circ lsc') * a \Mapsto l \\
(lsc * a \Mapsto [l]) \circ lsc' & \stackrel{\text{def}}{=} & (lsc \circ lsc') * a \Mapsto [l] \\
(lsc * a \Mapsto lc) \circ lsc' & \stackrel{\text{def}}{=} & \begin{cases}
(lsc \circ lsc') * a \Mapsto lc & \text{if } a \notin \mathrm{dom}(lsc') \\
(lsc \circ lsc'') * a \Mapsto lc[l/_] & \text{if } lsc' = lsc'' * a \Mapsto l \\
(lsc \circ lsc'') * a \Mapsto lc[lc'/_] & \text{if } lsc' = lsc'' * a \Mapsto lc'
\end{cases} \\
(lsc * a \Mapsto [lc]) \circ lsc' & \stackrel{\text{def}}{=} & \begin{cases}
(lsc \circ lsc') * a \Mapsto [lc] & \text{if } a \notin \mathrm{dom}(lsc') \\
(lsc \circ lsc'') * a \Mapsto [lc[l/_]] & \text{if } lsc' = lsc'' * a \Mapsto l \\
(lsc \circ lsc'') * a \Mapsto \left[lc[lc'/_]\right] & \text{if } lsc' = lsc'' * a \Mapsto lc'.
\end{cases}
\end{array}$$

Again, the result of the composition is undefined when either the right-hand side is badly formed or no case applies.

Definition 3.12 (List-Store Context Algebra). The *list-store context algebra*:

$$\mathcal{A}_{\mathbb{L}} \stackrel{\text{def}}{=} (\mathsf{C}_{\mathsf{LStore}}, \mathsf{LStore}, \circ, \bullet, \{\mathrm{emp}\}, \{\mathrm{emp}\})$$

is given by the above definitions.

Definition 3.13 (List Axiomatisation).
The *list axiomatisation*, $\mathrm{Ax}\,[\![(\cdot)]\!]_{\mathbb{L}} : \mathsf{Cmd}_{\mathbb{L}} \to \mathcal{P}(\mathcal{P}(\mathsf{Scope} \times \mathsf{LStore}) \times \mathcal{P}(\mathsf{Scope} \times \mathsf{LStore}))$

is defined as follows:

$$\mathrm{Ax}\,[\![\boldsymbol{x} := E.\texttt{getHead}()]\!]_{\mathbb{L}} \stackrel{\text{def}}{=} \left\{ \left(\begin{array}{c} (\boldsymbol{x} \Rightarrow v * \rho \times a \mapsto [w \cdot l])\,, \\ (\boldsymbol{x} \Rightarrow w * \rho \times a \mapsto [w \cdot l]) \end{array} \right) \;\middle|\; a = \mathcal{E}\,[\![E]\!]\,(\boldsymbol{x} \Rightarrow v * \rho) \right\} \cup \left\{ \left(\begin{array}{c} (\boldsymbol{x} \Rightarrow v * \rho \times a \mapsto [\varnothing])\,, \\ (\boldsymbol{x} \Rightarrow \boldsymbol{nil} * \rho \times a \mapsto [\varnothing]) \end{array} \right) \;\middle|\; a = \mathcal{E}\,[\![E]\!]\,(\boldsymbol{x} \Rightarrow v * \rho) \right\}$$

$$\mathrm{Ax}\,[\![\boldsymbol{x} := E.\texttt{getTail}()]\!]_{\mathbb{L}} \stackrel{\text{def}}{=} \left\{ \left(\begin{array}{c} (\boldsymbol{x} \Rightarrow v * \rho \times a \mapsto [l \cdot w])\,, \\ (\boldsymbol{x} \Rightarrow w * \rho \times a \mapsto [l \cdot w]) \end{array} \right) \;\middle|\; a = \mathcal{E}\,[\![E]\!]\,(\boldsymbol{x} \Rightarrow v * \rho) \right\} \cup \left\{ \left(\begin{array}{c} (\boldsymbol{x} \Rightarrow v * \rho \times a \mapsto [\varnothing])\,, \\ (\boldsymbol{x} \Rightarrow \boldsymbol{nil} * \rho \times a \mapsto [\varnothing]) \end{array} \right) \;\middle|\; a = \mathcal{E}\,[\![E]\!]\,(\boldsymbol{x} \Rightarrow v * \rho) \right\}$$

$$\mathrm{Ax}\,[\![\boldsymbol{x} := E_1.\texttt{getNext}(E_2)]\!]_{\mathbb{L}} \stackrel{\text{def}}{=} \left\{ \left(\begin{array}{c} (\boldsymbol{x} \Rightarrow v * \rho \times a \mapsto w \cdot u)\,, \\ (\boldsymbol{x} \Rightarrow u * \rho \times a \mapsto w \cdot u) \end{array} \right) \;\middle|\; \begin{array}{r} a = \mathcal{E}\,[\![E_1]\!]\,(\boldsymbol{x} \Rightarrow v * \rho) \text{ and} \\ w = \mathcal{E}\,[\![E_2]\!]\,(\boldsymbol{x} \Rightarrow v * \rho) \end{array} \right\} \cup \left\{ \left(\begin{array}{c} (\boldsymbol{x} \Rightarrow v * \rho \times a \mapsto [l \cdot w])\,, \\ (\boldsymbol{x} \Rightarrow \boldsymbol{nil} * \rho \times a \mapsto [l \cdot w]) \end{array} \right) \;\middle|\; \begin{array}{r} a = \mathcal{E}\,[\![E_1]\!]\,(\boldsymbol{x} \Rightarrow v * \rho) \text{ and} \\ w = \mathcal{E}\,[\![E_2]\!]\,(\boldsymbol{x} \Rightarrow v * \rho) \end{array} \right\}$$

$$\mathrm{Ax}\,[\![\boldsymbol{x} := E_1.\texttt{getPrev}(E_2)]\!]_{\mathbb{L}} \stackrel{\text{def}}{=} \left\{ \left(\begin{array}{c} (\boldsymbol{x} \Rightarrow v * \rho \times a \mapsto u \cdot w)\,, \\ (\boldsymbol{x} \Rightarrow u * \rho \times a \mapsto u \cdot w) \end{array} \right) \;\middle|\; \begin{array}{r} a = \mathcal{E}\,[\![E_1]\!]\,(\boldsymbol{x} \Rightarrow v * \rho) \text{ and} \\ w = \mathcal{E}\,[\![E_2]\!]\,(\boldsymbol{x} \Rightarrow v * \rho) \end{array} \right\} \cup \left\{ \left(\begin{array}{c} (\boldsymbol{x} \Rightarrow v * \rho \times a \mapsto [w \cdot l])\,, \\ (\boldsymbol{x} \Rightarrow \boldsymbol{nil} * \rho \times a \mapsto [w \cdot l]) \end{array} \right) \;\middle|\; \begin{array}{r} a = \mathcal{E}\,[\![E_1]\!]\,(\boldsymbol{x} \Rightarrow v * \rho) \text{ and} \\ w = \mathcal{E}\,[\![E_2]\!]\,(\boldsymbol{x} \Rightarrow v * \rho) \end{array} \right\}$$

$$\mathrm{Ax}\,[\![\boldsymbol{x} := E.\texttt{pop}()]\!]_{\mathbb{L}} \stackrel{\text{def}}{=} \left\{ \left(\begin{array}{c} (\boldsymbol{x} \Rightarrow v * \rho \times a \mapsto [w \cdot l])\,, \\ (\boldsymbol{x} \Rightarrow w * \rho \times a \mapsto [l]) \end{array} \right) \;\middle|\; a = \mathcal{E}\,[\![E]\!]\,(\boldsymbol{x} \Rightarrow v * \rho) \right\}$$

$$\mathrm{Ax}\,[\![E_1.\texttt{push}(E_2)]\!]_{\mathbb{L}} \stackrel{\text{def}}{=} \left\{ \left((\rho \times a \mapsto [l] \wedge v \notin l)\,, (\rho \times a \mapsto [v \cdot l]) \right) \;\middle|\; \begin{array}{r} a = \mathcal{E}\,[\![E_1]\!]\,\rho \text{ and} \\ v = \mathcal{E}\,[\![E_2]\!]\,\rho \end{array} \right\}$$

$$\mathrm{Ax}\,[\![E_1.\texttt{remove}(E_2)]\!]_{\mathbb{L}} \stackrel{\text{def}}{=} \left\{ \left((\rho \times a \mapsto v)\,, (\rho \times a \mapsto \varnothing) \right) \;\middle|\; a = \mathcal{E}\,[\![E_1]\!]\,\rho \text{ and } v = \mathcal{E}\,[\![E_2]\!]\,\rho \right\}$$

$$\mathrm{Ax}\,[\![E_1.\mathtt{insert}(E_2,E_3)]\!]_{\mathbb{L}} \stackrel{\text{def}}{=} \left\{ \left(\begin{array}{r} (\rho \times a \mapsto [l_1 \cdot v \cdot l_2] \wedge w \notin l_1 \cdot v \cdot l_2)\,, \\ (\rho \times a \mapsto [l_1 \cdot v \cdot w \cdot l_2]) \end{array} \right) \;\middle|\; \begin{array}{r} a = \mathcal{E}\,[\![E_1]\!]\,\rho \text{ and} \\ v = \mathcal{E}\,[\![E_2]\!]\,\rho \text{ and} \\ w = \mathcal{E}\,[\![E_3]\!]\,\rho \end{array} \right\}$$

$$\mathrm{Ax}\,[\![\boldsymbol{x} := \mathtt{newList}()]\!]_{\mathbb{L}} \stackrel{\text{def}}{=} \left\{ \left((\boldsymbol{x} \Rrightarrow - \times \mathrm{emp})\,, (\exists a.\, \boldsymbol{x} \Rrightarrow a \times a \mapsto [\varnothing]) \right) \right\}$$

$$\mathrm{Ax}\,[\![\mathtt{deleteList}(E)]\!]_{\mathbb{L}} \stackrel{\text{def}}{=} \left\{ \left((\rho \times a \mapsto [l])\,, (\rho \times \mathrm{emp}) \right) \;\middle|\; a = \mathcal{E}\,[\![E]\!]\,\rho \right\}$$

3.4. *Combining Abstract Modules*

It is often useful when programming to be able to make use of several modules. Just as it is natural to combine context algebras, as in Example 1(e), it is also natural to have a mechanism for combining abstract modules. The most intuitive approach is to allow arbitrary interleavings of their commands, whilst interpreting these commands over the product of their data-store context algebras. If the modules want to share any information, this must be done through the common variable store.

Definition 3.14 (Abstract Module Combination). Given abstract modules $\mathbb{A}_1 = (\mathsf{Cmd}_{\mathbb{A}_1}, \mathcal{A}_{\mathbb{A}_1}, \mathrm{Ax}\,[\![(\cdot)]\!]_{\mathbb{A}_1})$ and $\mathbb{A}_2 = (\mathsf{Cmd}_{\mathbb{A}_2}, \mathcal{A}_{\mathbb{A}_2}, \mathrm{Ax}\,[\![(\cdot)]\!]_{\mathbb{A}_2})$, their combination

$$\mathbb{A}_1 + \mathbb{A}_2 \stackrel{\text{def}}{=} (\mathsf{Cmd}_{\mathbb{A}_1} \oplus \mathsf{Cmd}_{\mathbb{A}_2}, \mathcal{A}_{\mathbb{A}_1} \times \mathcal{A}_{\mathbb{A}_2}, \mathrm{Ax}\,[\![(\cdot)]\!]_{\mathbb{A}_1+\mathbb{A}_2})$$

is an abstract module, where

— $\mathsf{Cmd}_{\mathbb{A}_1} \oplus \mathsf{Cmd}_{\mathbb{A}_2} \stackrel{\text{def}}{=} (\mathsf{Cmd}_{\mathbb{A}_1} \times \{1\}) \cup (\mathsf{Cmd}_{\mathbb{A}_2} \times \{2\})$ is the discriminated union of the command sets,
— $\mathcal{A}_{\mathbb{A}_1} \times \mathcal{A}_{\mathbb{A}_2}$ is the direct product of the context algebras, and
— $\mathrm{Ax}\,[\![(\cdot)]\!]_{\mathbb{A}_1+\mathbb{A}_2} : \mathsf{Cmd}_{\mathbb{A}_1} \oplus \mathsf{Cmd}_{\mathbb{A}_2} \to \mathcal{P}(\mathcal{P}(\mathsf{Scope} \times \mathsf{D}_{\mathbb{A}_1} \times \mathsf{D}_{\mathbb{A}_2}) \times \mathcal{P}(\mathsf{Scope} \times \mathsf{D}_{\mathbb{A}_1} \times \mathsf{D}_{\mathbb{A}_2}))$ is defined as

$$\begin{array}{rcl} \mathrm{Ax}\,[\![(\varphi,1)]\!]_{\mathbb{A}_1+\mathbb{A}_2} & \stackrel{\text{def}}{=} & \left\{ (\pi_1(P), \pi_1(Q) \;\middle|\; (P,Q) \in \mathrm{Ax}\,[\![\varphi]\!]_{\mathbb{A}_1} \right\} \\ \mathrm{Ax}\,[\![(\varphi,2)]\!]_{\mathbb{A}_1+\mathbb{A}_2} & \stackrel{\text{def}}{=} & \left\{ (\pi_2(P), \pi_2(Q) \;\middle|\; (P,Q) \in \mathrm{Ax}\,[\![\varphi]\!]_{\mathbb{A}_2} \right\} \end{array}$$

where

$$\begin{array}{rcl} \pi_1(P) & \stackrel{\text{def}}{=} & \{(\rho, \chi_1, o_2) \mid (\rho, \chi_1) \in P \text{ and } o_2 \in \mathbf{0}_2\} \\ \pi_2(P) & \stackrel{\text{def}}{=} & \{(\rho, o_1, \chi_2) \mid (\rho, \chi_2) \in P \text{ and } o_1 \in \mathbf{0}_1\}\,. \end{array}$$

Notation.
When the command sets $\mathsf{Cmd}_{\mathbb{A}_1}$ and $\mathsf{Cmd}_{\mathbb{A}_2}$ are disjoint, we will drop the tags when referring to the commands in the combined abstract module. When the tags are necessary, we will indicate them with an appropriately place subscript.

In §5.5 we shall use a combination of the heap module $\mathbb{H}$ and the list module $\mathbb{L}$ to provide am implementation of the tree module $\mathbb{T}$.

4. Module Translations

We now define what it means to correctly implement one module in terms of another, using *module translations*.

Definition 4.1 (Module Translation). A module translation $\tau : \mathbb{A} \to \mathbb{B}$ from abstract module $\mathbb{A}$ to abstract module $\mathbb{B}$ comprises:

— an *abstraction relation* $\alpha_\tau \subseteq \mathsf{D}_\mathbb{B} \times \mathsf{D}_\mathbb{A}$, and
— a *substitutive implementation function* $[\![(\cdot)]\!]_\tau : \mathcal{L}_\mathbb{A} \to \mathcal{L}_\mathbb{B}$ which uniformly substitutes each basic command of $\mathsf{Cmd}_\mathbb{A}$ with a call to a procedure written in $\mathcal{L}_\mathbb{B}$.

Notation.
The abstraction relation α_τ is lifted to a *predicate translation* $[\![(\cdot)]\!]_\tau : \mathcal{P}(\mathsf{State}_\mathbb{A}) \to \mathcal{P}(\mathsf{State}_\mathbb{B})$ as follows:

$$[\![P]\!] = \{(\rho, \chi_\mathbb{B}) \mid \text{there exists } \chi_\mathbb{A} \text{ s.t. } (\rho, \chi_\mathbb{A}) \in P \text{ and } \chi_\mathbb{B}\ \alpha_\tau\ \chi_\mathbb{A}\}\,.$$

When the translation τ is implicit from context, the subscripts on the abstraction relation, implementation function and predicate translation may be dropped.

In the context of a translation $\tau : \mathbb{A} \to \mathbb{B}$, $\mathbb{A}$ is called the *abstract* or *high-level* module and $\mathbb{B}$ is called the *concrete* or *low-level* module. (Of course, there is no reason why a module should not be abstract with respect to one translation and concrete with respect to another, or even both the abstract and concrete module with respect to a single translation.)

Definition 4.2 (Sound Module Translation). A module translation $\tau : \mathbb{A} \to \mathbb{B}$ is said to be sound, if for all $P, Q \in \mathcal{P}(\mathsf{State}_\mathbb{A})$ and $\mathbb{C} \in \mathcal{L}_\mathbb{A}$,

$$\vdash_\mathbb{A} \{P\}\ \mathbb{C}\ \{Q\} \quad \Longrightarrow \quad \vdash_\mathbb{B} \{[\![P]\!]_\tau\}\ [\![\mathbb{C}]\!]_\tau\ \{[\![Q]\!]_\tau\}\,.$$

Intuitively, a sound module translation appears to be a reasonable correctness condition for a module implementation: everything that can be proved about the abstract module also holds for its implementation. There are, however, a few caveats.

Firstly, since we have elected to work with partial correctness Hoare triples, it is acceptable for an implementation to simply loop forever. If termination guarantees are required, they could either be made separately or a logic based on total correctness could be used. We have chosen to work with partial correctness for simplicity and on the basis that partial correctness is generally used in the separation logic and context logic literature (Ishtiaq and O'Hearn, 2001; Reynolds, 2002; Calcagno et al., 2005).

Secondly, it is possible for the abstraction relation to lose information. For instance, if all predicates were unsatisfiable under translation then it would be possible to soundly implement every abstract command with `skip`; such an implementation is useless. One way of mitigating this would be to consider a set of *initial predicates* that must be satisfiable under translation. A triple whose precondition is such an initial predicate is then meaningful under translation, since it does not hold vacuously. A more stringent approach would be to require the abstraction relation α_τ to be surjective, and therefore

every satisfiable predicate to be satisfiable under translation. This condition is, however, not met by one of the natural implementations considered here.

In this section, we present two techniques for constructing sound module translations: locality-preserving translations and locality-breaking translations. *Locality-preserving translations*, discussed in §5, closely preserve the structure of the abstract module's context algebra through the translation, which leads to an elegant inductive proof transformation from the abstract to the concrete. In particular, context application at the abstract level corresponds to context application at the concrete level, and so the abstract frame rule is transformed to the concrete frame rule. *Locality-breaking translations*, discussed in §6, on the other hand do not necessarily preserve the structure of the abstract module's context algebra. Even so, certain properties of the proof theory can be used to simplify the problem of establishing the soundness of such a translation.

4.1. *Modularity*

An important property of module translations is that they are composable: given translations $\tau_1 : \mathbb{A}_1 \to \mathbb{A}_2$ and $\tau_2 : \mathbb{A}_2 \to \mathbb{A}_3$, the translation $\tau_2 \circ \tau_1 : \mathbb{A}_1 \to \mathbb{A}_3$ can be defined in the natural fashion. If its constituent translations are sound then so is the composition. Therefore, it is possible to construct module translations stepwise.

A translation $\tau : \mathbb{A}_1 \to \mathbb{A}_2$ can be naturally lifted to a translation $\tau + \mathbb{B} : \mathbb{A}_1 + \mathbb{B} \to \mathbb{A}_2 + \mathbb{B}$, for any module $\mathbb{B}$. If τ is a sound translation, we might also expect $\tau + \mathbb{B}$ to be sound. However, it is not obvious that this is the case in general. The techniques for constructing sound translations in this paper do, however, admit such a lifting. This is because they transform high-level proofs to low-level proofs in a fashion that preserves any additional module component. Being able to combine translations for independent modules means that these techniques are modular.

5. Locality-Preserving Translations

Sometimes, there is a close correspondence between the locality exhibited by a high-level module and the locality exhibited by the low-level implementation of the module. In this section, we expand on this intuition and formalise the concept of a *locality-preserving translation*. In §5.1, we establish that locality-preserving translations give sound module translations, and in §5.4, §5.5 and §5.6 we give locality-preserving translations $\tau_1 : \mathbb{T} \to \mathbb{H}$, $\tau_2 : \mathbb{T} \to \mathbb{H} + \mathbb{L}$ and $\tau_3 : \mathbb{H} + \mathbb{H} \to \mathbb{H}$.

So what exactly does it mean for there to be a correspondence between locality at the high level and locality at the low level? Consider Fig. 3, which depicts a typical tree from the tree module $\mathbb{T}$ (a), together with possible representations of that tree in the heap module $\mathbb{H}$ (b), and in the combined heap-and-list module $\mathbb{H} + \mathbb{L}$ (c). In Fig. 3(b), each tree node is represented by a memory block comprising four pointer fields (depicted by a circle with outgoing arrows) which record the addresses of the memory blocks representing the left sibling, parent, right sibling and first child. Where there is no such node (for example, when a node has no children) the pointer field holds the ***nil*** value (depicted by the absence of an arrow). In Fig. 3(c), each tree node is represented by a

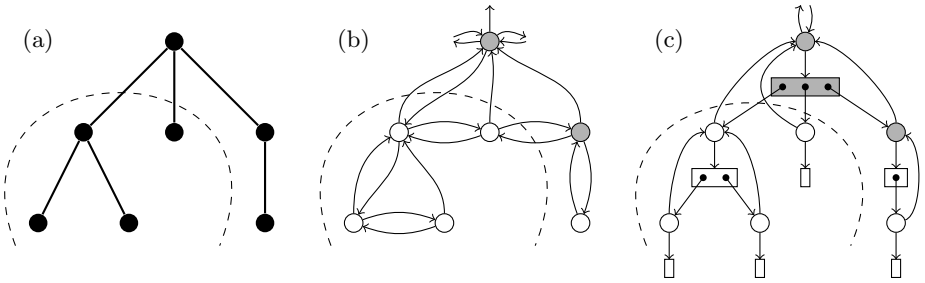

Fig. 3. An abstract tree from $\mathbb{T}$ (a), and representations of the tree in $\mathbb{H}$ (b), and in $\mathbb{H} + \mathbb{L}$ (c).

memory block that comprises a pointer to the node's parent and a pointer to the child list of the node. The child list (depicted by a box with dots for each value in the list) is a list of pointers to the node's children.

What these examples exhibit is a simple inductive transformation from the abstract data structure to its concrete representations. This suggests the possibility of a simple inductive transformation from high-level *proofs* to low-level proofs: in particular, it should be possible to transform high-level frames into low-level frames. Such a transformation may well be said to preserve locality.

Essential to constructing such a transformation is that the concrete representation of abstract data preserves the abstract application structure. In particular, if we split the abstract data structure into a context and subdata, its concrete representation can also be split into the representations of this context and subdata. In Fig. 3, the dashed line indicates such a splitting of the abstract tree, and the corresponding splittings of the two representations. In general, for all contexts c and abstract data structures χ, we would like their intuitive representations, denoted $\langle\!\langle c \rangle\!\rangle$ and $\langle\!\langle \chi \rangle\!\rangle$, to satisfy the *application preservation property* $\langle\!\langle c \bullet \chi \rangle\!\rangle \equiv \langle\!\langle c \rangle\!\rangle \bullet \langle\!\langle \chi \rangle\!\rangle$.

The reality is, however, more subtle than this intuition, since the concrete representations of context and data must mesh together correctly. In the example representations we have considered, this means that the pointers between the context and subdata must link up correctly. Thus, while an abstract context is agnostic to the data that is placed in its hole, the representation of the context needs to "know" some information about the representation of the data, and vice-versa. Thus the representations of context and data need to be parametrised by *interfaces* which record this "knowledge" that the context and data need about each other. Specifically, the representation of abstract data structure χ is given by the concrete predicate $\langle\!\langle \chi \rangle\!\rangle^I$, and the representation of abstract context c is given by the concrete predicate $\langle\!\langle c \rangle\!\rangle^I_{I'}$. The context representation requires two interfaces: the I' between it and the data in its hole and the I between it and its own surrounding context. Taking interfaces into account, the *application preservation property* is now, essentially,

$$\langle\!\langle c \bullet \chi \rangle\!\rangle^I \equiv \exists I'.\, \langle\!\langle c \rangle\!\rangle^I_{I'} \bullet \langle\!\langle \chi \rangle\!\rangle^{I'}.$$

The interface is divided into two parts: the knowledge that a context representation needs about the data to be put in its hole is called the *in-interface*, while the knowledge that a data representation needs about its surrounding context is called the *out-interface*. In Fig. 3 the dashed lines depict a splitting of the abstract tree, and its representations, into a context and a subtree. In Fig. 3(b), the in-interface consists of the addresses of the left- and right-most nodes at the top level of the subtree, while the out-interface consists of the addresses of the nodes immediately left, above and to the right of the context hole. In Fig. 3(c), the in-interface consists of the addresses of all of the nodes at the top level of the subtree, while the out-interface consists of the address of the parent node of the context hole. Typically, as in these examples, the interfaces are a collection of pointers, which are identified in the diagrams by the arrows that cross the dividing line between context and data. The direction of the arrow determines whether it belongs to the in- or out-interface.

Having established an application-preserving representation of data, we aim to transform high-level proofs about an abstract program into corresponding low-level proofs about its implementation, by simply replacing the high-level predicates with their low-level representations. Most of the proof rules should transform simply to their low-level counterparts; application preservation should allow us to transform the Frame rule; and to deal with Axiom we simply need to prove that the implementations of the basic commands satisfy the low-level representations of their specifications. However, consider the operation of disposing the subtree indicated in Fig. 3. At the abstract level, it is clear that the resource required to run the command is just the subtree that is to be deleted. This is reflected in the axiom:

$$\mathrm{Ax}\, [\![\texttt{deleteTree}(E)]\!]_{\mathbb{T}} \stackrel{\text{def}}{=} \left\{ \big((\rho \times \mathbf{a}[t]) \,, (\rho \times \mathbf{0}) \big) \;\middle|\; \mathcal{E}\, [\![E]\!]\, (\boldsymbol{x} \Rightarrow v * \rho) = \mathbf{a} \right\}$$

Yet in both implementations, something more than the representation of the subtree is required: for the heap implementation in Fig. 3(b), the pointers from the context into the deleted subtree must be updated; for the heap-and-list implementation in Fig. 3(c), the pointers to the subtree's top-level nodes must be removed from their parent's child list. In both cases, the low-level footprint of dispose is larger than the intuitive representation. The axiom for dispose cannot therefore simply be translated.

We could decide that our chosen representation is unsuitable. In our two tree implementations however, the representations seem pretty intuitive. Instead, we demonstrate that it is possible to repair our existing approach. We do this by introducing the concept of *crust*, a predicate that corresponds to the minimal extra resource (taken from the surrounding context) that is required by the command implementations. In Fig. 3, the crust for each subtree representation is depicted by the shaded portions of the context representation. For the heap implementation in Fig. 3(b), this is the nodes (in the example, the parent and right sibling) that have pointers into the subtree; for the heap-and-list implementation in Fig. 3(c), this is the parent node, its child list and its other children. In general, the crust is represented by the predicate $\Cap_I^F$, parametrised by an interface I and some additional parameter F that together fully determine it. (Since the crust corresponds to some portion of the surrounding context, some information about that context is contained within the crust; F is the part of this information that is not

already provided by I. In the heap implementation of trees, for example, F provides all the values of pointers in the nodes belonging to the crust which are not in I.)

We must now ensure that the *axiom correctness property* holds with this extra crust. This means we must check that the implementations of the basic commands satisfy the representations of their specifications, with the addition of crust. Loosely:

$$\vdash \left\{\exists in.\, \Cap{}{}{F}_{in,out} \bullet \langle\!\langle P \rangle\!\rangle^{in,out}\right\}\ [\![\varphi]\!]\ \left\{\exists in.\, \Cap{}{}{F}_{in,out} \bullet \langle\!\langle Q \rangle\!\rangle^{in,out}\right\}$$

for every $(P, Q) \in \mathrm{Ax}\,[\![\varphi]\!]$.

With crust now added to the representation of data, the application preservation property no longer immediately provides a way to translate high-level frames into low-level frames. If we simply took the low-level frame to be the representation of the high-level frame plus its outer crust, this would duplicate the data's crust. This *inner crust* must therefore be removed from the frame before it is applied. For example, consider again disposing the subtree indicated in Fig. 3. In the heap implementation of Fig. 3(b), the command operates on the representation of the subtree (as indicated by the dashed line) plus the crust (as indicated by the shaded nodes). If the context is added by frame at the high-level, what is added at the low-level is the representation of the context (and its outer crust) *minus* the inner crust (the shaded nodes). The possibility of removing the inner crust in this way is established by the *crust inclusion property*, which (in part) states that the representation of a context together with its outer crust contains the inner crust. Loosely:

$$\left(\exists in'.\, \Cap{}{}{F}_{in',out'} \circ \langle\!\langle c \rangle\!\rangle_I^{in',out'}\right) \equiv K \circ \Cap{}{}{F'}_I .$$

This crust inclusion property enables us to transform a high-level frame into a low-level frame, and hence construct a *locality-preserving* module translation. We introduce crust to deal with the fact that the footprint of the low-level operations overlaps with the representation of the context. Hence we say that such a module translation provides a *fiction of disjointness*.

Having fleshed out the intuition behind locality-preserving translations, we now introduce their formal definition. We first define the concept of *pre-locality-preserving translations*, which have the appropriate form, and then restrict locality-preserving translations to being those that exhibit the properties of application preservation, crust inclusion and axiom correctness. We prove a general result that locality-preserving translations are sound module translations.

Definition 5.1 (Pre-Locality-Preserving Translation). A *pre-locality-preserving translation* $\tau : \mathbb{A} \to \mathbb{B}$ comprises:

— a set of *in-interfaces* $\mathcal{I}_{\mathrm{in}}$ and a set of *out-interfaces* $\mathcal{I}_{\mathrm{out}}$, whose Cartesian product constitutes the set of *interfaces* $\mathcal{I} = \mathcal{I}_{\mathrm{in}} \times \mathcal{I}_{\mathrm{out}}$;
— a *data representation function* $\langle\!\langle (\cdot) \rangle\!\rangle^{(\cdot)} : \mathsf{D}_{\mathbb{A}} \times \mathcal{I} \to \mathcal{P}(\mathsf{D}_{\mathbb{B}})$;
— a *context representation function* $\langle\!\langle (\cdot) \rangle\!\rangle^{(\cdot)}_{(\cdot)} : \mathsf{C}_{\mathbb{A}} \times \mathcal{I} \times \mathcal{I} \to \mathcal{P}(\mathsf{C}_{\mathbb{B}})$;
— a set of *crust parameters* $\mathcal{F}$;
— a crust predicate $\Cap{}{}{F}_I \in \mathcal{P}(\mathsf{C}_{\mathbb{B}})$, parametrised by interface $I \in \mathcal{I}$ and crust parameter $F \in \mathcal{F}$; and

— a *substitutive implementation function* $[\![(\cdot)]\!]_\tau : \mathcal{L}_\mathbb{A} \to \mathcal{L}_\mathbb{B}$.

We construct a module translation from a pre-locality-preserving translation by defining the abstraction relation in terms of the data representation function and the crust predicate. For any given $out \in \mathcal{I}_{\text{out}}$ and $F \in \mathcal{F}$, the abstraction relation α is defined to be

$$\alpha = \left\{ (\chi_\mathbb{B}, \chi_\mathbb{A}) \mid \chi_\mathbb{B} \in \exists in.\, \Cap{}^{F}_{in,out} \bullet \langle\!\langle \chi_\mathbb{A} \rangle\!\rangle^{in,out} \right\}.$$

Frequently, there will be a natural choice of *out* and F, but in general any choice is permissible.

We define an *intermediate predicate translation* $(\!|(\cdot)|\!)^{(\cdot)}$, which is closely related to the predicate translation $[\![(\cdot)]\!]$, except that it is explicitly parametrised by the choice of *out* and F.

Definition 5.2 (Intermediate Predicate Translation). Given a pre-locality-preserving translation, the *intermediate predicate translation function*

$$(\!|(\cdot)|\!)^{(\cdot)} : \mathcal{P}(\mathsf{State}_\mathbb{A}) \times (\mathcal{I}_{\text{out}} \times \mathcal{F}) \to \mathcal{P}(\mathsf{State}_\mathbb{B})$$

is defined as:

$$(\!|P|\!)^{out,F} \stackrel{\text{def}}{=} \bigvee_{(\rho,\chi_\mathbb{A}) \in P} \{\rho\} \times \left(\exists in.\, \Cap{}^{F}_{in,out} \bullet \langle\!\langle \chi_\mathbb{A} \rangle\!\rangle^{in,out} \right).$$

For a module translation defined by a pre-locality-preserving translation with respect to $out \in \mathcal{I}_{\text{out}}$ and $F \in \mathcal{F}$, the predicate translation can be expressed simply in terms of the intermediate predicate translation: $[\![P]\!] = (\!|P|\!)^{out,F}$.

Pre-locality-preserving translations do not embody the intuition of what it means for a module translation to soundly preserve locality. This is reserved for locality-preserving translations, which require the following three properties.

Property 1 (Application Preservation). Context application is preserved by the representation functions. That is, for all $c \in \mathsf{C}_\mathbb{A}$, $\chi \in \mathsf{D}_\mathbb{A}$ and $I \in \mathcal{I}$,

$$\langle\!\langle c \bullet_\mathbb{A} \chi \rangle\!\rangle^I \equiv \exists I'.\, \langle\!\langle c \rangle\!\rangle^I_{I'} \bullet_\mathbb{B} \langle\!\langle \chi \rangle\!\rangle^{I'}.$$

Property 2 (Crust Inclusion). For all $out, out' \in \mathcal{I}_{\text{out}}$, $F \in \mathcal{F}$ and $c \in \mathsf{C}_\mathbb{A}$, there exist $K \in \mathcal{P}(\mathsf{C}_\mathbb{B})$ and $F' \in \mathcal{F}$ such that, for all $in \in \mathcal{I}_{\text{in}}$,

$$\left(\exists in'.\, \Cap{}^{F}_{in',out'} \circ \langle\!\langle c \rangle\!\rangle^{in',out'}_{in,out} \right) \equiv K \circ \Cap{}^{F'}_{in,out}.$$

Property 3 (Axiom Correctness). For all $\varphi \in \mathsf{Cmd}_\mathbb{A}$, $(P, Q) \in \mathrm{Ax}\,[\![\varphi]\!]_\mathbb{A}$, $out \in \mathcal{I}_{\text{out}}$ and $F \in \mathcal{F}$,

$$\vdash_\mathbb{B} \left\{ (\!|P|\!)^{out,F} \right\} [\![\varphi]\!] \left\{ (\!|Q|\!)^{out,F} \right\}.$$

Definition 5.3 (Locality Preserving Translation). A *locality-preserving translation* is a pre-locality-preserving translation that satisfies Properties 1, 2 and 3.

Remark. The crust inclusion property is stronger than just asserting that inner crust is contained within the context plus its outer crust. This is to allow for the fact that

a command may modify the crust, and we must be able to use this modified crust to reconstitute the representation of the same context. However, the crust is only ever altered with respect to the in-interface, so it is only necessary that K in Property 2 be independent of the choice of in.

Theorem 5.1 (Soundness of Locality-Preserving Translations). A locality-preserving translation is a sound module translation (for any fixed choice of $out \in \mathcal{I}_{\text{out}}$ and $F \in \mathcal{F}$).

5.1. *Proof of Soundness of Locality-Preserving Translations*

In this section, assume that $\tau : \mathbb{A} \to \mathbb{B}$ is a locality preserving translation. The following proposition is sufficient to establish that τ gives rise to a sound module translation.

Proposition 5.2. For all $out \in \mathcal{I}_{\text{out}}$ and $F \in \mathcal{F}$,and for all $P, K \in \mathcal{P}(\mathsf{State}_{\mathbb{A}})$ and $\mathbb{C} \in \mathcal{L}_{\mathbb{A}}$,

$$\Gamma \vdash_{\mathbb{A}} \{P\}\ \mathbb{C}\ \{Q\} \implies [\![\Gamma]\!] \vdash_{\mathbb{B}} \left\{ (\!|P|\!)^{out,F} \right\}\ \mathbb{C}\ \left\{ (\!|Q|\!)^{out,F} \right\},$$

where

$$[\![\Gamma]\!] = \left\{ \mathtt{f} : (\!|P'|\!)^{out',F'} \rightarrowtail (\!|Q'|\!)^{out',F'} \;\middle|\; \begin{array}{l} (\mathtt{f} : P' \rightarrowtail Q') \in \Gamma \text{ and} \\ out' \in \mathcal{I}_{\text{out}} \text{ and } F' \in \mathcal{F} \end{array} \right\}.$$

Before embarking on the proof of this proposition, two auxiliary lemmata are required. The following lemma gives and alternative characterisation of the crust inclusion property.

Lemma 5.3 (Crust Inclusion II). For all $K \in \mathcal{P}(\mathsf{C}_{\mathbb{A}})$, $in \in \mathcal{I}_{\text{in}}$, $out, out' \in \mathcal{I}_{\text{out}}$ and $F \in \mathcal{F}$,

$$\left(\exists in'.\, \Cap{}{}{F}_{in',out'} \circ \langle\!\langle K \rangle\!\rangle_{in,out}^{in',out'}\right) \subseteq \exists F'.\, \left(\forall in''.\, \Cap{}{}{F'}_{in'',out} \multimap \left(\exists in'.\, \Cap{}{}{F}_{in',out'} \circ \langle\!\langle K \rangle\!\rangle_{in'',out}^{in',out'}\right)\right) \circ \Cap{}{}{F'}_{in,out},$$

where

$$\langle\!\langle K \rangle\!\rangle_{I}^{I'} = \bigvee_{c \in K} \langle\!\langle c \rangle\!\rangle_{I}^{I'}$$

is the point-wise lift of the context representation function to predicates.

Note that the converse of this property is trivially true, hence the entailment holds in both directions.

Proof. Fix arbitrary $K \in \mathcal{P}(\mathsf{C}_{\mathbb{A}})$, $in \in \mathcal{I}_{\text{in}}$, $out, out' \in \mathcal{I}_{\text{out}}$ and $F \in \mathcal{F}$. Fix $c' \in \mathsf{C}_{\mathbb{A}}$ with

$$c' \in \left(\exists in'.\, \Cap{}{}{F}_{in',out'} \circ \langle\!\langle K \rangle\!\rangle_{in,out}^{in',out'}\right) \equiv \bigvee_{c \in K} \left(\exists in'.\, \Cap{}{}{F}_{in',out'} \circ \langle\!\langle c \rangle\!\rangle_{in,out}^{in',out'}\right).$$

There exists $c'' \in K$ such that

$$c' \in \left(\exists in'.\, \Cap{}{}{F}_{in',out'} \circ \langle\!\langle c'' \rangle\!\rangle_{in,out}^{in',out'}\right).$$

By the Crust Inclusion Property, there exist $K' \in \mathcal{P}(\mathsf{C}_{\mathbb{B}})$ and $F' \in \mathcal{F}$ such that, for all $in'' \in \mathcal{I}_{\text{in}}$,

$$\left(\exists in'.\, \mathbb{m}^{F}_{in',out'} \circ \langle\!\langle c'' \rangle\!\rangle^{in',out'}_{in'',out}\right) \equiv K' \circ \mathbb{m}^{F'}_{in'',out}. \tag{1}$$

Hence, $c' \in K' \circ \mathbb{m}^{F'}_{in,out}$, and so there are $c_1 \in K'$ and $c_2 \in \mathbb{m}^{F'}_{in,out}$ with $c' = c_1 \circ c_2$. Fix $in'' \in \mathcal{I}_{\text{in}}$ and $c_2' \in \mathbb{m}^{F'}_{in'',out}$. Since $c_1 \circ \circ_2 \in K' \circ \mathbb{m}^{F'}_{in'',out}$, it follows by (1) that

$$c_1 \circ c_2' \in \left(\exists in'.\, \mathbb{m}^{F}_{in',out'} \circ \langle\!\langle c'' \rangle\!\rangle^{in',out'}_{in'',out}\right) \subseteq \exists in'.\, \mathbb{m}^{F}_{in',out'} \circ \langle\!\langle K \rangle\!\rangle^{in',out'}_{in'',out}.$$

The choice of c_2' was arbitrary, and so

$$\text{for all } c_2', c'' \in \mathsf{C}_{\mathbb{B}},\ c_2' \in \mathbb{m}^{F}_{in'',out} \text{ and } c'' = c_1 \circ c_2' \implies c'' \in \exists in'.\, \mathbb{m}^{F}_{in',out'} \bullet \langle\!\langle K \rangle\!\rangle^{in',out'}_{in'',out}.$$

Hence

$$c_1 \in \mathbb{m}^{F}_{in'',out} \multimap \left(\exists in'.\, \mathbb{m}^{F}_{in',out'} \bullet \langle\!\langle K \rangle\!\rangle^{in',out'}_{in'',out}\right)$$

and since the choice of in'' was arbitrary,

$$c_1 \in \forall in''.\, \mathbb{m}^{F}_{in'',out} \multimap \left(\exists in'.\, \mathbb{m}^{F}_{in',out'} \bullet \langle\!\langle K \rangle\!\rangle^{in',out'}_{in'',out}\right)$$

Since $c' = c_1 \circ c_2$,

$$c' \in \exists F'.\, \left(\forall in''.\, \mathbb{m}^{F}_{in'',out} \multimap \left(\exists in'.\, \mathbb{m}^{F}_{in',out'} \bullet \langle\!\langle K \rangle\!\rangle^{in',out'}_{in'',out}\right)\right) \circ \mathbb{m}^{F'}_{in,out}.$$

Since the choice of c' was arbitrary, it follows that

$$\left(\exists in'.\, \mathbb{m}^{F}_{in',out'} \circ \langle\!\langle K \rangle\!\rangle^{in',out'}_{in,out}\right) \subseteq \exists F'.\, \left(\forall in''.\, \mathbb{m}^{F'}_{in'',out} \multimap \left(\exists in'.\, \mathbb{m}^{F}_{in',out'} \circ \langle\!\langle K \rangle\!\rangle^{in',out'}_{in'',out}\right)\right) \circ \mathbb{m}^{F'}_{in,out}$$

as required. □

We define the notion of an *intermediate frame translation*, which translates abstract frames (context predicates) to concrete frames. The intermediate frame translation combines the context representation with the outer crust, but removes the inner crust. When the frame as applied at the concrete level, the inner crust will already be present: removing the inner crust from the context is necessary to avoid duplication.

Definition 5.4 (Intermediate Frame Translation). The *intermediate frame translation function*

$$(\!|(\cdot)|\!)^{(\cdot)}_{(\cdot)} : \mathcal{P}(\mathsf{C}_{\mathsf{State}_{\mathbb{A}}}) \times (\mathcal{I}_{\text{out}} \times \mathcal{F}) \times (\mathcal{I}_{\text{out}} \times \mathcal{F}) \to \mathcal{P}(\mathsf{C}_{\mathsf{State}_{\mathbb{B}}})$$

is defined as:

$$(\!|K|\!)^{out,F}_{out',F'} = \bigvee_{(\rho,c_{\mathbb{A}}) \in K} \{\rho\} \times \left(\forall in'.\, \mathbb{m}^{F}_{in',out'} \multimap \left(\exists in.\, \mathbb{m}^{F}_{in,out} \circ \langle\!\langle c_{\mathbb{A}} \rangle\!\rangle^{in,out}_{in',out'}\right)\right).$$

The fact that the intermediate frame translation produces concrete frames that exactly correspond to abstract frames is critical for dealing with the FRAME rule in the proof of Proposition 5.2. This fact is captured formally in the following lemma.

Lemma 5.4 (Application Preservation II). For all $K \in \mathcal{P}(\mathsf{C}_{\mathsf{State}_{\mathbb{A}}})$, all $P \in \mathcal{P}(\mathsf{State}_{\mathbb{A}})$, $out \in \mathcal{I}_{\mathrm{out}}$ and $F \in \mathcal{F}$,

$$(\!|K \bullet_{\mathbb{A}} P|\!)^{out,F} \equiv \exists out', F'.\, (\!|K|\!)^{out,F}_{out',F'} \bullet_{\mathbb{B}} (\!|P|\!)^{out',F'}.$$

Proof.

$$(\!|K \bullet P|\!)^{out,F} \equiv \bigvee_{\substack{(\rho',c_{\mathbb{A}})\in K\\(\rho,\chi_{\mathbb{A}})\in P}} \{\rho' * \rho\} \times \left(\exists in.\, \Cap{}{}{F}_{in,out} \bullet \langle\!\langle c_{\mathbb{A}} \bullet \chi_{\mathbb{A}} \rangle\!\rangle^{in,out}\right)$$

(Property 1)

$$\equiv \bigvee_{\substack{(\rho',c_{\mathbb{A}})\in K\\(\rho,\chi_{\mathbb{A}})\in P}} \{\rho' * \rho\} \times \left(\exists in.\, \Cap{}{}{F}_{in,out} \bullet \exists in', out'.\, \langle\!\langle c_{\mathbb{A}} \rangle\!\rangle^{in,out}_{in',out'} \bullet \langle\!\langle \chi_{\mathbb{A}} \rangle\!\rangle^{in',out'}\right)$$

$$\equiv \bigvee_{\substack{(\rho',c_{\mathbb{A}})\in K\\(\rho,\chi_{\mathbb{A}})\in P}} \{\rho' * \rho\} \times \left(\exists in', out'.\, \exists in.\, \Cap{}{}{F}_{in,out} \circ \langle\!\langle c_{\mathbb{A}} \rangle\!\rangle^{in,out}_{in',out'} \bullet \langle\!\langle \chi_{\mathbb{A}} \rangle\!\rangle^{in',out'}\right)$$

(Lemma 5.3)

$$\equiv \bigvee_{\substack{(\rho',c_{\mathbb{A}})\in K\\(\rho,\chi_{\mathbb{A}})\in P}} \{\rho' * \rho\} \times \left(\begin{array}{l} \exists in', out'.\, \exists F'.\\ \left(\forall in''.\, \Cap{}{}{F'}_{in'',out'} \multimap \left(\exists in.\, \Cap{}{}{F}_{in,out} \circ \langle\!\langle c_{\mathbb{A}} \rangle\!\rangle^{in,out}_{in'',out'}\right)\right)\\ \qquad\qquad \circ\, \Cap{}{}{F'}_{in',out'} \bullet \langle\!\langle \chi_{\mathbb{A}} \rangle\!\rangle^{in',out'} \end{array}\right)$$

$$\equiv \exists out', F'.$$

$$\left(\bigvee_{(\rho',c_{\mathbb{A}})\in K} \{\rho'\} \times \left(\forall in''.\, \Cap{}{}{F'}_{in'',out'} \multimap \left(\exists in.\, \Cap{}{}{F}_{in,out} \circ \langle\!\langle c_{\mathbb{A}} \rangle\!\rangle^{in,out}_{in'',out'}\right)\right)\right)$$

$$\bullet \left(\bigvee_{(\rho,\chi_{\mathbb{A}})\in P} \{\rho\} \times \left(\exists in'.\, \Cap{}{}{F'}_{in',out'} \bullet \langle\!\langle \chi_{\mathbb{A}} \rangle\!\rangle^{in',out'}\right)\right)$$

$$\equiv \exists out', F'.\, (\!|K|\!)^{out,F}_{out',F'} \bullet (\!|P|\!)^{out',F'}.$$

□

The proof of Proposition 5.2 inductively transforms a proof in $\mathbb{A}$ into a proof in $\mathbb{B}$.

Proof. The proof is by induction on the structure of the proof of $\vdash_{\mathbb{A}} \{P\}\ \mathbb{C}\ \{Q\}$, considering the cases for the last rule applied in the proof. Assume as the inductive hypothesis that the translated premises have proofs in $\mathbb{B}$. We show how to derive a proof of the translated conclusions from these translated premises. (We omit the procedure specification environment when it plays no role in the derivation.)

Fix arbitrary $out \in \mathcal{I}_{\mathrm{out}}$, $F \in \mathcal{F}$.

Axiom case:

This case is immediate by the Axiom Correctness Property (Property 3).

FRAME case:

$$\dfrac{\dfrac{\dfrac{\text{for all } out', F',\ \left\{(\!|P|\!)^{out',F'}\right\}\ \mathbb{C}\ \left\{(\!|Q|\!)^{out',F'}\right\}}{\text{for all } out', F',\ \begin{array}{c}\left\{(\!|K|\!)^{out,F}_{out',F'} \bullet (\!|P|\!)^{out',F'}\right\}\\ \mathbb{C}\\ \left\{(\!|K|\!)^{out,F}_{out',F'} \bullet (\!|Q|\!)^{out',F'}\right\}\end{array}}\ \textsc{Disj}}{\begin{array}{c}\left\{\exists out', F'.\, (\!|K|\!)^{out,F}_{out',F'} \bullet (\!|P|\!)^{out',F'}\right\}\\ \mathbb{C}\\ \left\{\exists out', F'.\, (\!|K|\!)^{out,F}_{out',F'} \bullet (\!|Q|\!)^{out',F'}\right\}\end{array}}\ \textsc{Frame}}{\left\{(\!|K \bullet P|\!)^{out,F}\right\}\ \mathbb{C}\ \left\{(\!|K \bullet Q|\!)^{out,F}\right\}}\ \text{Lemma 5.4}$$

CONS case:

$$\dfrac{\dfrac{P \subseteq P'}{(\!|P|\!)^{out,F} \subseteq (\!|P'|\!)^{out,F}}\quad \begin{array}{c}\left\{(\!|P'|\!)^{out,F}\right\}\\ \mathbb{C}\\ \left\{(\!|Q'|\!)^{out,F}\right\}\end{array}\quad \dfrac{Q' \subseteq Q}{(\!|Q'|\!)^{out,F} \subseteq (\!|Q|\!)^{out,F}}}{\left\{(\!|P|\!)^{out,F}\right\}\ \mathbb{C}\ \left\{(\!|Q|\!)^{out,F}\right\}}\ \textsc{Cons}$$

DISJ case:

$$\dfrac{\dfrac{\text{for all } i \in I,\ \left\{(\!|P_i|\!)^{out,F}\right\}\ \mathbb{C}\ \left\{(\!|Q_i|\!)^{out,F}\right\}}{\left\{\bigvee_{i\in I} (\!|P_i|\!)^{out,F}\right\}\ \mathbb{C}\ \left\{\bigvee_{i\in I} (\!|Q_i|\!)^{out,F}\right\}}\ \textsc{Disj}}{\left\{(\!|\bigvee_{i\in I} P_i|\!)^{out,F}\right\}\ \mathbb{C}\ \left\{(\!|\bigvee_{i\in I} Q_i|\!)^{out,F}\right\}}$$

PDEF case:

$$\dfrac{\dfrac{\text{for all } (\mathtt{f}_i : \mathsf{P} \rightarrowtail \mathsf{Q}) \in \Gamma,\ out' \in \mathcal{I}_{\text{out}}, F' \in \mathcal{F},\ [\![\Gamma', \Gamma]\!] \vdash_{\mathbb{B}} \begin{array}{c}\left\{\left(\!\left|\ \exists \overrightarrow{v}.\, \{\overrightarrow{\boldsymbol{x}_i} \Rightarrow \overrightarrow{v} * \overrightarrow{\boldsymbol{r}_i} \Rightarrow -\} \times \mathsf{P}(\overrightarrow{v})\ \right|\!\right)^{out',F'}\right\}\\ \mathbb{C}_i\\ \left\{\left(\!\left|\ \exists \overrightarrow{w}.\, \{\overrightarrow{\boldsymbol{x}_i} \Rightarrow - * \overrightarrow{\boldsymbol{r}_i} \Rightarrow \overrightarrow{w}\} \times \mathsf{Q}(\overrightarrow{w})\ \right|\!\right)^{out',F'}\right\}\end{array}}{\text{for all } (\mathtt{f}_i : \mathsf{P} \rightarrowtail \mathsf{Q}) \in \Gamma,\ out' \in \mathcal{I}_{\text{out}}, F' \in \mathcal{F},\ [\![\Gamma', \Gamma]\!] \vdash_{\mathbb{B}} \begin{array}{c}\left\{\ \exists \overrightarrow{v}.\, \{\overrightarrow{\boldsymbol{x}_i} \Rightarrow \overrightarrow{v} * \overrightarrow{\boldsymbol{r}_i} \Rightarrow -\} \times (\!|\mathsf{P}(\overrightarrow{v})|\!)^{out',F'}\ \right\}\\ \mathbb{C}_i\\ \left\{\ \exists \overrightarrow{w}.\, \{\overrightarrow{\boldsymbol{x}_i} \Rightarrow - * \overrightarrow{\boldsymbol{r}_i} \Rightarrow \overrightarrow{w}\} \times (\!|\mathsf{Q}(\overrightarrow{w})|\!)^{out',F'}\ \right\}\end{array}}}{\text{for all } (\mathtt{f}_i : \mathsf{P} \rightarrowtail \mathsf{Q}) \in [\![\Gamma]\!],\ [\![\Gamma', \Gamma]\!] \vdash_{\mathbb{B}} \begin{array}{c}\{\exists \overrightarrow{v}.\, \{\overrightarrow{\boldsymbol{x}_i} \Rightarrow \overrightarrow{v} * \overrightarrow{\boldsymbol{r}_i} \Rightarrow -\} \times \mathsf{P}(\overrightarrow{v})\}\\ \mathbb{C}_i\\ \{\exists \overrightarrow{w}.\, \{\overrightarrow{\boldsymbol{x}_i} \Rightarrow - * \overrightarrow{\boldsymbol{r}_i} \Rightarrow \overrightarrow{w}\} \times \mathsf{Q}(\overrightarrow{w})\}\end{array}}\ (\star)$$

$$\frac{(\star)\quad [\![\Gamma', \Gamma]\!] \vdash_{\mathbb{B}} \left\{ (\!| P |\!)^{out,F} \right\}\ \mathbb{C}\ \left\{ (\!| Q |\!)^{out,F} \right\}}{[\![\Gamma']\!] \vdash_{\mathbb{B}} \texttt{procs}\ \overrightarrow{r_1} := \mathtt{f}_1(\overrightarrow{x_1})\ \{\mathbb{C}_1\}, \ldots, \overrightarrow{r_k} := \mathtt{f}_k(\overrightarrow{x_k})\ \{\mathbb{C}_k\}\ \texttt{in}\ \mathbb{C}\ \left\{ (\!| R |\!)^{out,F} \right\}}\ \textsc{PDef}$$

The two further premises of the PDEF rule, which are not shown in the above derivation, are easily dispatched since $[\![(\cdot)]\!]$ preserves the procedure names in a procedure specification environment.

The cases for the remaining rules follow by the point-wise and variable-preserving nature of the translation functions. □

This completes the proof of Theorem 5.1.

5.2. *Eliminating Crust*

Our theory includes the explicit concept of crust in order to provide additional resource that was necessary for the low-level operations, but not included in the representation of the abstract resource. However, in general it is possible to choose a different representation that avoids the need for crust. In fact, such a choice is embodied in a key element of the soundness proof, namely the intermediate translation functions (Definitions 5.2 and 5.4).

The intermediate translation functions essentially add the outer crust to representations of data structures and contexts while removing the inner crust from representations of contexts. Because of the crust inclusion and application preservation properties of the original representations, this modified representation also preserves application (Lemma 5.4). (Note that the interface set for this modified representation is $\mathcal{I}_{\text{out}} \times \mathcal{F}$, which can all be considered as the out part.) This modified representation incorporates all of the resource required at the low level by its very construction, and so no further crust is required.

5.3. *Including the Conjunction Rule*

If we wish to add the conjunction rule to the locality-preserving theory, we can add a case to the proof of Proposition 5.2. The CONJ rule can be dealt with in the same fashion as the DISJ rule, provided that $(\!| (\cdot) |\!)^{(\cdot)}$ distributes over conjunctions. Together, the following two conditions are sufficient to establish this:

— for all $\chi, \chi' \in \mathsf{D}$ with $\chi \neq \chi'$, and all $I \in \mathcal{I}$, $\langle\!\langle \chi \rangle\!\rangle^I \wedge \langle\!\langle \chi' \rangle\!\rangle^I \equiv \emptyset$; and
— for all $out \in \mathcal{I}_{\text{out}}$ and $F \in \mathcal{F}$, the predicate $\bigvee \left\{ \Subset^F_{in,out} \;\middle|\; in \in \mathcal{I}_{\text{in}} \right\}$ is precise.

Remark. It is not a coincidence that these conditions are similar to those that prevent a command from behaving angelically given in §2: in both cases, the conditions are constraining the predicate transformers corresponding to the abstraction relation or command to being *conjunctive*.

5.4. *Module Translation:* $\tau_1 : \mathbb{T} \to \mathbb{H}$

We now present a locality-preserving translation τ_1 from the tree module $\mathbb{T}$ into the heap module $\mathbb{H}$. This translation represents each tree node $\mathbf{a}$ as a block of four cells in the heap, $\mathbf{a} \mapsto l, u, d, r$, which contain pointers to the node's left sibling (l), parent (u), first child (d) and right sibling (r). This is the representation illustrated in Fig. 3(b).

As expected, the in-interface consists of the addresses of the left- and right-most nodes at the root level of a tree, while the out-interface consists of the addresses of the tree's parent node and of the nodes immediately adjacent to the tree on the left and right.

Note that for the empty tree $\varnothing$, the addresses constituting the in-interface are not simply $\boldsymbol{nil}$; rather the "address of the left-most node" should actually be the address of the node immediately adjacent to the tree on the right and the "address of the right-most node" should be the address of the node adjacent on the left. If this were not the case, it would disrupt the continuous list of nodes. On the other hand, if the nodes described in the out-interface do not exist, their addresses will be $\boldsymbol{nil}$.

For contexts, addresses in the inner interface and outer interface can be the same. This is particularly evident in the case of the context hole, $-$, for which the two interfaces must match exactly.

Definition 5.5 ($\tau_1 : \mathbb{T} \to \mathbb{H}$). The pre-locality-preserving translation $\tau_1 : \mathbb{T} \to \mathbb{H}$ is constructed as follows:

— the in-interfaces $\mathcal{I}_{\text{in}} = (\mathsf{Addr}_{\boldsymbol{nil}})^2$ are pairs of addresses;
— the out-interfaces $\mathcal{I}_{\text{out}} = (\mathsf{Addr}_{\boldsymbol{nil}})^3$ are triples of addresses;
— the data representation function is defined inductively by

$$\begin{aligned}
\langle\!\langle \varnothing \rangle\!\rangle^{(i,j),(l,u,r)} &\stackrel{\text{def}}{=} \text{emp} \wedge i = r \wedge j = l \\
\langle\!\langle \mathbf{a}[t] \rangle\!\rangle^{(i,j),(l,u,r)} &\stackrel{\text{def}}{=} \exists i', j'.\, \mathbf{a} \mapsto l, u, i', r * \langle\!\langle t \rangle\!\rangle^{(i',j'),(\boldsymbol{nil},\mathbf{a},\boldsymbol{nil})} \wedge i = j = \mathbf{a} \\
\langle\!\langle t_1 \otimes t_2 \rangle\!\rangle^{(i,j),(l,u,r)} &\stackrel{\text{def}}{=} \exists i', j'.\, \langle\!\langle t_1 \rangle\!\rangle^{(i,j'),(l,u,i')} * \langle\!\langle t_2 \rangle\!\rangle^{(i',j),(j',u,r)};
\end{aligned}$$

— the context representation function is defined inductively by

$$\begin{aligned}
\langle\!\langle - \rangle\!\rangle^{I}_{I'} &\stackrel{\text{def}}{=} \text{emp} \wedge I = I' \\
\langle\!\langle \mathbf{a}[c] \rangle\!\rangle^{(i,j),(l,u,r)}_{I'} &\stackrel{\text{def}}{=} \exists i', j'.\, \mathbf{a} \mapsto l, u, i', r * \langle\!\langle c \rangle\!\rangle^{(i',j'),(\boldsymbol{nil},\mathbf{a},\boldsymbol{nil})}_{I'} \wedge i = j = \mathbf{a} \\
\langle\!\langle t \otimes c \rangle\!\rangle^{(i,j),(l,u,r)}_{I'} &\stackrel{\text{def}}{=} \exists i', j'.\, \langle\!\langle t \rangle\!\rangle^{(i,j'),(l,u,i')} * \langle\!\langle c \rangle\!\rangle^{(i',j),(j',u,r)}_{I'} \\
\langle\!\langle c \otimes t \rangle\!\rangle^{(i,j),(l,u,r)}_{I'} &\stackrel{\text{def}}{=} \exists i', j'.\, \langle\!\langle c \rangle\!\rangle^{(i,j'),(l,u,i')}_{I'} * \langle\!\langle t \rangle\!\rangle^{(i',j),(j',u,r)};
\end{aligned}$$

— the crust parameters $\mathcal{F} = (\mathsf{Addr}_{\boldsymbol{nil}})^7$ are 7-tuples of addresses;
— the crust predicate is defined as

$$\begin{aligned}
\Subset^{\vec{f}}_{(i,j),(l,u,r)} \stackrel{\text{def}}{=} (l \mapsto f_1, u, f_2, i \vee (l = \boldsymbol{nil} \wedge (u \mapsto f_3, f_4, i, f_5 \vee (u = \boldsymbol{nil} \wedge \text{emp})))) \\
* (r \mapsto j, u, f_6, f_7 \vee (r = \boldsymbol{nil} \wedge \text{emp})); \text{ and}
\end{aligned}$$

— the substitutive implementation function is given by replacing each tree-module command with a call to the correspondingly-named procedure given in Fig. 4.

$$E.\texttt{left} \stackrel{\text{def}}{=} E$$
$$E.\texttt{up} \stackrel{\text{def}}{=} E + 1$$
$$E.\texttt{down} \stackrel{\text{def}}{=} E + 2$$
$$E.\texttt{right} \stackrel{\text{def}}{=} E + 3$$
$$n := \texttt{newNode}() \stackrel{\text{def}}{=} n := \texttt{alloc}(4)$$
$$\texttt{disposeNode}(E) \stackrel{\text{def}}{=} \texttt{dispose}(E, 4)$$

```
m := getUp(n) {
  m := [n.up]
}

m := getLast(n) {
  local x in
    m := [n.down];
    if m ≠ nil then
      x := [m.right];
      while x ≠ nil do
        m := x;
        x := [m.right]
}

newNodeAfter(n) {
  local x, y, z in
    y := [n.right];
    z := [n.up];
    z := newNode();
    [x.left] := n;
    [x.up] := z;
    [x.down] := nil;
    [x.right] := y;
    [n.right] := x;
    if y ≠ nil then
      [y.left] := x
}

m := getFirst(n) {
  m := [n.down]
}

m := getRight(n) {
  m := [n.right]
}

m := getLeft(n) {
  m := [n.left]
}

deleteTree(n) {
  local x, y, z, w in
    x := [n.right];
    y := [n.left];
    z := [n.up];
    w := [n.down];
    call disposeForest(w);
    if x ≠ nil then
      [x.left] := y   ;
    if y ≠ nil then
      [y.right] := x
    else
      if z ≠ nil then
        [z.down] := x
}

disposeForest(n) {
  local r, d in
    if n ≠ nil then
      r := [n.right];
      call disposeForest(r);
      d := [n.down];
      call disposeForest(d);
      disposeNode(n)
}
```

Fig. 4. Procedures for the heap-based implementation of trees

Theorem 5.5 (Soundness of τ_1). The pre-locality-preserving translation τ_1 is a locality-preserving translation.

5.4.1. *Soundness of* $\tau_1 : \mathbb{T} \to \mathbb{H}$

Lemma 5.6 (τ_1 Application Preservation). For all $c \in \mathsf{C}_{\mathsf{Tree}}$, $t \in \mathsf{Tree}$ and $I \in \mathcal{I}$,

$$\langle\!\langle c \bullet t \rangle\!\rangle^I \equiv \exists I'.\, \langle\!\langle c \rangle\!\rangle^I_{I'} \bullet \langle\!\langle t \rangle\!\rangle^{I'}.$$

Proof. The proof is by induction on the structure of the context c, assuming some fixed $t \in \mathsf{Tree}$.

$c = \_$ case:

$$\begin{aligned}\exists I'.\, \langle\!\langle \_ \rangle\!\rangle^I_{I'} \bullet \langle\!\langle t \rangle\!\rangle^{I'} &\equiv \exists I'.\, I = I' \wedge \langle\!\langle t \rangle\!\rangle^{I'} \\ &\equiv \langle\!\langle t \rangle\!\rangle^I \\ &\equiv \langle\!\langle \_ \bullet t \rangle\!\rangle^I.\end{aligned}$$

$c = \mathbf{a}[c']$ case:

$$\begin{aligned}&\exists I'.\, \langle\!\langle \mathbf{a}[c'] \rangle\!\rangle^{(i,j),(l,u,r)}_{I'} \bullet \langle\!\langle t \rangle\!\rangle^{I'} \\ &\quad\equiv \exists I'.\, \left(\exists i', j'.\, \mathbf{a} \mapsto l, u, i', r * \langle\!\langle c' \rangle\!\rangle^{(i',j'),(\mathbf{nil},\mathbf{a},\mathbf{nil})}_{I'} \wedge i = j = \mathbf{a}\right) \bullet \langle\!\langle t \rangle\!\rangle^{I'} \\ &\quad\equiv \exists i', j'.\, \mathbf{a} \mapsto l, u, i', r * \left(\exists I'.\, \langle\!\langle c' \rangle\!\rangle^{(i',k'),(\mathbf{nil},\mathbf{a},\mathbf{nil})}_{I'} \bullet \langle\!\langle t \rangle\!\rangle^{I'}\right) \wedge i = j = \mathbf{a} \\ &\quad\equiv \exists i', j'.\, \mathbf{a} \mapsto l, u, i', r * \langle\!\langle c' \bullet t \rangle\!\rangle^{(i',k'),(\mathbf{nil},\mathbf{a},\mathbf{nil})} \wedge i = j = \mathbf{a} \\ &\quad\equiv \langle\!\langle \mathbf{a}[c' \bullet t] \rangle\!\rangle^{(i,j),(l,u,r)} \\ &\quad\equiv \langle\!\langle \mathbf{a}[c'] \bullet t \rangle\!\rangle^{(i,j),(l,u,r)}.\end{aligned}$$

$c = c' \otimes t'$ case:

$$\begin{aligned}&\exists I'.\, \langle\!\langle c' \otimes t' \rangle\!\rangle^{(i,j),(l,u,r)}_{I'} \bullet \langle\!\langle t \rangle\!\rangle^{I'} \\ &\quad\equiv \exists I'.\, \left(\exists i', j'.\, \langle\!\langle c' \rangle\!\rangle^{(i,j'),(l,u,i')}_{I'} * \langle\!\langle t' \rangle\!\rangle^{(i',j),(j',u,r)}\right) \bullet \langle\!\langle t \rangle\!\rangle^{I'} \\ &\quad\equiv \exists i', j'.\, \left(\exists I'.\, \langle\!\langle c' \rangle\!\rangle^{(i,j'),(l,u,i')}_{I'} \bullet \langle\!\langle t \rangle\!\rangle^{I'}\right) * \langle\!\langle t' \rangle\!\rangle^{(i',j),(j',u,r)} \\ &\quad\equiv \exists i', j'.\, \langle\!\langle c' \bullet t \rangle\!\rangle^{(i,j'),(l,u,i')} * \langle\!\langle t' \rangle\!\rangle^{(i',j),(j',u,r)} \\ &\quad\equiv \langle\!\langle (c' \bullet t) \otimes t' \rangle\!\rangle^{(i,j),(l,u,r)} \\ &\quad\equiv \langle\!\langle (c' \otimes t') \bullet t \rangle\!\rangle^{(i,j),(l,u,r)}.\end{aligned}$$

The case where $c = t' \otimes c'$ follows the same pattern as the $c' \otimes t'$ case. □

The following lemma is used to prove the crust inclusion property. It essentially captures that a given tree and out-interface uniquely determine an in-interface. When the tree is non-empty, the in-interface is the identifiers of the left-most and right-most nodes at the top level of the tree; when the tree is empty, the out-interface provides the correct identifiers.

Lemma 5.7. For all $out \in \mathcal{I}_{\text{out}}$ and $t \in \mathsf{Tree}$, there exists some $in \in \mathcal{I}_{\text{in}}$ such that

$$\langle\!\langle t \rangle\!\rangle^{in,out} \equiv \exists in'.\, \langle\!\langle t \rangle\!\rangle^{in',out}.$$

The proof is by a straightforward induction on the structure of the tree t.

Lemma 5.8 (τ_1 Crust Inclusion). For all $out, out' \in \mathcal{I}_{\text{out}}$, $F \in \mathcal{F}$ and $c \in \mathsf{C}_{\mathsf{Tree}}$, there exist $K \in \mathcal{P}(\mathsf{Heap})$ and $F' \in \mathcal{F}$ such that, for all $in \in \mathcal{I}_{\text{in}}$,

$$\left(\exists in'.\, \Cap{F}_{in',out'} \circ \langle\!\langle c \rangle\!\rangle^{in',out'}_{in,out}\right) \equiv K \circ \Cap{F'}_{in,out}.$$

Proof. The proof is by induction on the structure of the context c, assuming some fixed $out, out' \in \mathcal{I}_{\text{out}}$ and $F \in \mathcal{F}$.

$c = \_$ case:
Choose $K = \text{emp}$ if $out' = out$ and $K = \text{False}$ otherwise. If $out' \neq out$ then both sides of the equation are equivalent to False; otherwise, observe that

$$\begin{aligned}\exists in'.\, \Cap{F}_{in',out'} \circ \langle\!\langle \_ \rangle\!\rangle^{in',out'}_{in,out} &\equiv \Cap{F}_{in,out} \\ &\equiv K \circ \Cap{F}_{in,out}.\end{aligned}$$

$c = \mathbf{a}[c']$ case:
Let $out'' = (\boldsymbol{nil}, \mathbf{a}, \boldsymbol{nil})$ and $F'' = (\boldsymbol{nil}, \boldsymbol{nil}, l, u, r, \boldsymbol{nil}, \boldsymbol{nil})$, where $(l, u, r) = out'$. By the inductive hypothesis, there exist K' and F' such that, for all in,

$$\exists i', j'.\, \Cap{F''}_{(i',j'),out''} \circ \langle\!\langle c' \rangle\!\rangle^{(i',j'),out''}_{in,out} \equiv K' \circ \Cap{F'}_{in,out}.$$

Choose $K = \Cap{F}_{(\mathbf{a},\mathbf{a}),out'} * K'$; choose F' as given above. Observe that, for all in,

$$\begin{aligned}&\exists in'.\, \Cap{F}_{in',out'} \circ \langle\!\langle c \rangle\!\rangle^{in',out'}_{in,out} \\ &\qquad\equiv \exists in'.\, \Cap{F}_{in',out'} * \exists i', j'.\, \mathbf{a} \mapsto l, u, d, r * \langle\!\langle c' \rangle\!\rangle^{(i',j'),(\boldsymbol{nil},\mathbf{a},\boldsymbol{nil})}_{I'} \wedge in' = (\mathbf{a}, \mathbf{a}) \\ &\qquad\equiv \Cap{F}_{(\mathbf{a},\mathbf{a}),out'} * \exists i', j'.\, \Cap{F''}_{(i',j'),out''} \circ \langle\!\langle c' \rangle\!\rangle^{(i',j'),out''}_{in,out} \\ &\qquad\equiv \Cap{F}_{(\mathbf{a},\mathbf{a}),out'} * K' \circ \Cap{F'}_{in,out} \\ &\qquad\equiv K \circ \Cap{F'}_{in,out}.\end{aligned}$$

$c = c' \otimes t$ case:
We can assume that $t \neq \varnothing$, since otherwise $c = c'$ and the result holds by the inductive assumption. Thus $t = \mathbf{a}[t_1] \otimes t_2$ for some $\mathbf{a}, t_1, t_2$. In order to apply the inductive hypothesis, we must determine the crust for the c' part of c. The left or parent node part of the crust will be the same as for c; the right node will be $\mathbf{a}$.

Suppose that $out' = (l, u, r)$ and $F = (f_1, \ldots, f_7)$. Let f'_6 be such that, for some j''',

$$\langle\!\langle t_1 \rangle\!\rangle^{(f'_6,j'''),(\boldsymbol{nil},\mathbf{a},\boldsymbol{nil})} \equiv \exists in'''.\, \langle\!\langle t_1 \rangle\!\rangle^{in''',(\boldsymbol{nil},\mathbf{a},\boldsymbol{nil})}.$$

Let f'_7 be such that, for some j,

$$\langle\!\langle t_2 \rangle\!\rangle^{(f'_7,j),(\mathbf{a},u,r)} \equiv \exists in''.\, \langle\!\langle t_2 \rangle\!\rangle^{in'',(\mathbf{a},u,r)}.$$

Let $F' = (f_1, \ldots, f_5, f'_6, f'_7)$. By the inductive hypothesis, let K' and F'' be such that,

for all in,

$$\exists i, j'.\, \Cap{}{}{F'}{(i,j'),(l,u,\mathbf{a})} \circ \Ltrans{c'}{(i,j'),(l,u,\mathbf{a})}{in,out} \equiv K' \circ \Cap{}{}{F''}{in,out}$$

Now, by construction, for arbitrary in,

$$\begin{aligned}
&\exists i, j.\, \Cap{}{}{F}{(i,j),out'} \circ \Ltrans{c' \otimes t}{(i,j),(l,u,r)}{in,out} \\
&\quad\equiv \exists i, j.\, \Cap{}{}{F}{(i,j),out'} \circ \exists i', j'.\, \Ltrans{c'}{(i,j'),(l,u,i')}{in,out} * \Ltrans{\mathbf{a}[t_1] \otimes t_2}{(i',j),(j',u,r)} \\
&\quad\equiv \exists i, j.\, \Cap{}{}{F}{(i,j),out'} \circ \exists i', j'.\, \Ltrans{c'}{(i,j'),(l,u,i')}{in,out} * \\
&\qquad\qquad \exists i'', j''.\, \Ltrans{\mathbf{a}[t_1]}{(i',j''),(j',u,i'')} * \Ltrans{t_2}{(i'',j),(j'',u,r)} \\
&\quad\equiv \exists i, j.\, \Cap{}{}{F}{(i,j),out'} \circ \exists j'.\, \Ltrans{c'}{(i,j'),(l,u,\mathbf{a})}{in,out} * \\
&\qquad\qquad \exists i'', i''', j'''.\, \mathbf{a} \mapsto j', u, i''', i'' * \Ltrans{t_1}{(i''',j'''),(\mathbf{nil},\mathbf{a},\mathbf{nil})} * \Ltrans{t_2}{(i'',j),(\mathbf{a},u,r)} \\
&\quad\equiv \exists i, j.\, \Cap{}{}{F}{(i,j),out'} \circ \exists j'.\, \Ltrans{c'}{(i,j'),(l,u,\mathbf{a})}{in,out} * \\
&\qquad\qquad \exists j'''.\, \mathbf{a} \mapsto j', u, f_6', f_7' * \Ltrans{t_1}{(f_6',j'''),(\mathbf{nil},\mathbf{a},\mathbf{nil})} * \Ltrans{t_2}{(f_7',j),(\mathbf{a},u,r)} \\
&\quad\equiv \exists i, j, j'.\, (l \mapsto f_1, u, f_2, i \vee (l = \mathbf{nil} \wedge (u \mapsto f_3, f_4, i, f_5 \vee (u = \mathbf{nil} \wedge \mathrm{emp})))) \\
&\qquad\qquad * (r \mapsto j, u, f_6, f_7 \vee (r = \mathbf{nil} \wedge \mathrm{emp})) \circ \Ltrans{c'}{(i,j'),(l,u,\mathbf{a})}{in,out} * \\
&\qquad\qquad\quad \mathbf{a} \mapsto j', u, f_6', f_7' * \exists j'''.\, \Ltrans{t_1}{(f_6',j'''),(\mathbf{nil},\mathbf{a},\mathbf{nil})} * \Ltrans{t_2}{(f_7',j),(\mathbf{a},u,r)} \\
&\quad\equiv \exists i, j'.\, (l \mapsto f_1, u, f_2, i \vee (l = \mathbf{nil} \wedge (u \mapsto f_3, f_4, i, f_5 \vee (u = \mathbf{nil} \wedge \mathrm{emp})))) \\
&\qquad\qquad * \mathbf{a} \mapsto j', u, f_6', f_7' \circ \Ltrans{c'}{(i,j'),(l,u,\mathbf{a})}{in,out} \\
&\qquad\qquad * \exists j, j''.\, (r \mapsto j, u, f_6, f_7 \vee (r = \mathbf{nil} \wedge \mathrm{emp})) * \\
&\qquad\qquad\quad \Ltrans{t_1}{(f_6',j'''),(\mathbf{nil},\mathbf{a},\mathbf{nil})} * \Ltrans{t_2}{(f_7',j),(\mathbf{a},u,r)} \\
&\quad\equiv \exists i, j'.\, \Cap{}{}{F'}{(i,j'),(l,u,\mathbf{a})} \circ \Ltrans{c'}{(i,j'),(l,u,\mathbf{a})}{in,out} \\
&\qquad\qquad * \exists j, j''.\, (r \mapsto j, u, f_6, f_7 \vee (r = \mathbf{nil} \wedge \mathrm{emp})) * \\
&\qquad\qquad\quad \Ltrans{t_1}{(f_6',j'''),(\mathbf{nil},\mathbf{a},\mathbf{nil})} * \Ltrans{t_2}{(f_7',j),(\mathbf{a},u,r)} \\
&\quad\equiv K' * \exists j, j''.\, (r \mapsto j, u, f_6, f_7 \vee (r = \mathbf{nil} \wedge \mathrm{emp})) * \\
&\qquad\qquad \Ltrans{t_1}{(f_6',j'''),(\mathbf{nil},\mathbf{a},\mathbf{nil})} * \Ltrans{t_2}{(f_7',j),(\mathbf{a},u,r)} \circ \Cap{}{}{F''}{in,out}
\end{aligned}$$

Let

$$\begin{aligned}
K \equiv K' * \exists j, j''.\, &(r \mapsto j, u, f_6, f_7 \vee (r = \mathbf{nil} \wedge \mathrm{emp})) * \\
&\Ltrans{t_1}{(f_6',j'''),(\mathbf{nil},\mathbf{a},\mathbf{nil})} * \Ltrans{t_2}{(f_7',j),(\mathbf{a},u,r)}.
\end{aligned}$$

K and F'' therefore have the desired property.

The case where $c = t \otimes c'$ is similar. □

Lemma 5.9. For all $\varphi \in \mathsf{Cmd}_{\mathbb{T}}$, $(P, Q) \in \mathrm{Ax}\,[\![\varphi]\!]_{\mathbb{T}}$, $out \in \mathcal{I}_{\mathrm{out}}$ and $F \in \mathcal{F}$,

$$\vdash_{\mathbb{H}} \left\{ (\!| P |\!)^{out,F} \right\}\ [\![\varphi]\!]\ \left\{ (\!| Q |\!)^{out,F} \right\}.$$

We omit the full proof details here, which can be found in (Dinsdale-Young et al., 2010b). Figs. 5 and 6 show proof outlines for the `getLast` axioms. The two cases cover when the node whose last child is to be fetched has and does not have children. The correctness of the axioms for `getLast` follow from these proofs by the procedure definition and procedure call rules.

This completes the proof of the soundness of the translation $\tau_1 : \mathbb{T} \to \mathbb{H}$ (Theorem 5.5).

5.5. *Module Translation:* $\tau_2 : \mathbb{T} \to \mathbb{H} + \mathbb{L}$

In this section, we give the formal definition of a locality-preserving translation τ_2 from the tree module $\mathbb{T}$ into the combination of the heap and list modules $\mathbb{H} + \mathbb{L}$.

Notation.
A number of notational conventions are used to simplify predicates over $\mathcal{A}_{\mathbb{H}} + \mathcal{A}_{\mathbb{L}}$. Pure heap predicates are implicitly lifted to the combined domain with the list-store component emp; list-store predicates are lifted similarly. The operator $*$ on the combined domain is the product of the $*$ operators from each of the two domains.

Definition 5.6 ($\tau_2 : \mathbb{T} \to \mathbb{H}+\mathbb{L}$). The pre-locality-preserving translation $\tau_2 : \mathbb{T} \to \mathbb{H}+\mathbb{L}$ is constructed as follows:

— the in-interfaces $\mathcal{I}_{\text{in}} = (\mathsf{Addr})^*$ are sequences of addresses;
— the out-interfaces $\mathcal{I}_{\text{out}} = \mathsf{Addr}$ are addresses;
— the data representation function is defined inductively by

$$\begin{array}{rcl} \langle\!\langle \varnothing \rangle\!\rangle^{in,out} & \stackrel{\text{def}}{=} & \text{emp} \wedge in = \varnothing \\ \langle\!\langle \mathbf{a}[t] \rangle\!\rangle^{in,out} & \stackrel{\text{def}}{=} & in = \mathbf{a} \wedge \exists a, l.\, \mathbf{a} \mapsto out, a * a \Rrightarrow [l] * \langle\!\langle t \rangle\!\rangle^{l,\mathbf{a}} \\ \langle\!\langle t_1 \otimes t_2 \rangle\!\rangle^{in,out} & \stackrel{\text{def}}{=} & \exists l_1, l_2.\, (in = l_1 \cdot l_2) \wedge \langle\!\langle t_1 \rangle\!\rangle^{l_1,out} * \langle\!\langle t_2 \rangle\!\rangle^{l_2,out}; \end{array}$$

— the context representation function is defined inductively by

$$\begin{array}{rcl} \langle\!\langle - \rangle\!\rangle^{in,out}_{in',out'} & \stackrel{\text{def}}{=} & \text{emp} \wedge (in = in') \wedge (out = out') \\ \langle\!\langle \mathbf{a}[c] \rangle\!\rangle^{in,out}_{I'} & \stackrel{\text{def}}{=} & in = \mathbf{a} \wedge \exists a, l.\, \mathbf{a} \mapsto out, a * a \Rrightarrow [l] * \langle\!\langle c \rangle\!\rangle^{l,\mathbf{a}}_{I'} \\ \langle\!\langle c \otimes t \rangle\!\rangle^{in,out}_{I'} & \stackrel{\text{def}}{=} & \exists l_1, l_2.\, (in = l_1 \cdot l_2) \wedge \langle\!\langle c \rangle\!\rangle^{l_1,out}_{I'} * \langle\!\langle t \rangle\!\rangle^{l_2,out} \\ \langle\!\langle t \otimes c \rangle\!\rangle^{in,out}_{I'} & \stackrel{\text{def}}{=} & \exists l_1, l_2.\, (in = l_1 \cdot l_2) \wedge \langle\!\langle t \rangle\!\rangle^{l_1,out} * \langle\!\langle c \rangle\!\rangle^{l_2,out}_{I'}; \end{array}$$

— the crust parameters $\mathcal{F} = (\mathsf{Addr})^* \times (\mathsf{Addr})^* \times \mathsf{Addr}_{\boldsymbol{nil}}$ are tuples of two sequences of addresses and a further address or ***nil***;
— the crust predicate is defined as

$$\Cap^{l_1,l_2,a}_{in,out} \stackrel{\text{def}}{=} \exists a'.\, out \mapsto a, a' * a' \Rrightarrow [l_1 \cdot in \cdot l_2] * \prod_{\mathbf{a} \in l_1 \cdot l_2}^{*} \mathbf{a} \mapsto out; \text{ and}$$

— the substitutive implementation function is given by replacing each tree-module command with a call to the correspondingly-named procedure given in Fig. 7.

Theorem 5.10 (Soundness of τ_2). The pre-locality-preserving translation τ_2 is a locality-preserving translation.

$$
\begin{array}{l}
m := \texttt{getLast}(n)\ \{ \\
\quad \left\{ n \Rrightarrow \mathbf{a} * m \Rrightarrow - \times \Cap^{F}_{(\mathbf{a},\mathbf{a}),(l,u,r)} * \langle\!\langle \mathbf{a}[t_1 \otimes \mathbf{b}[t_2]] \rangle\!\rangle^{(\mathbf{a},\mathbf{a}),(l,u,r)} \right\} \\
\quad \texttt{local}\ x\ \texttt{in} \\
\qquad \left\{ \exists i, j.\, n \Rrightarrow \mathbf{a} * m \Rrightarrow - * x \Rrightarrow - \times \Cap^{F}_{(\mathbf{a},\mathbf{a}),(l,u,r)} * \mathbf{a} \mapsto l, u, i, r * \langle\!\langle t_1 \otimes \mathbf{b}[t_2] \rangle\!\rangle^{(i,j),(\mathit{nil},\mathbf{a},\mathit{nil})} \right\} \\
\qquad m := [n.\texttt{down}]\,; \\
\qquad \left\{ \exists i, j.\, n \Rrightarrow \mathbf{a} * m \Rrightarrow i * x \Rrightarrow - \times \Cap^{F}_{(\mathbf{a},\mathbf{a}),(l,u,r)} * \mathbf{a} \mapsto l, u, i, r * \langle\!\langle t_1 \otimes \mathbf{b}[t_2] \rangle\!\rangle^{(i,j),(\mathit{nil},\mathbf{a},\mathit{nil})} \right\} \\
\qquad\quad \left\{ m \Rrightarrow i * x \Rrightarrow - \times \langle\!\langle t_1 \otimes \mathbf{b}[t_2] \rangle\!\rangle^{(i,j),(\mathit{nil},\mathbf{a},\mathit{nil})} \right\} \\
\qquad\quad // \ t_1 \otimes \mathbf{b}[t_2] \neq \varnothing \implies i \neq \mathit{nil} \\
\qquad\quad \texttt{if}\ m \neq \mathit{nil}\ \texttt{then} \\
\qquad\qquad \left\{ \begin{array}{l} \exists d, x, t', t''.\, (i[t'] \otimes t'' = t_1 \otimes \mathbf{b}[t_2]) \wedge m \Rrightarrow i * x \Rrightarrow - \\ \qquad \times i \mapsto \mathit{nil}, \mathbf{a}, d, x * \langle\!\langle t' \rangle\!\rangle^{(d,-),(\mathit{nil},i,\mathit{nil})} * \langle\!\langle t'' \rangle\!\rangle^{(x,j),(i,\mathbf{a},\mathit{nil})} \end{array} \right\} \\
\qquad\qquad x := [m.\texttt{right}]\,; \\
\qquad\qquad \left\{ \begin{array}{l} \exists m, j_1, d, x, t, t', t''.\, (t \otimes m[t'] \otimes t'' = t_1 \otimes \mathbf{b}[t_2]) \wedge m \Rrightarrow m * x \Rrightarrow x \\ \qquad \times \langle\!\langle t \rangle\!\rangle^{(i,j_1),(\mathit{nil},\mathbf{a},m)} * m \mapsto j_1, \mathbf{a}, d, x * \langle\!\langle t' \rangle\!\rangle^{(d,-),(\mathit{nil},m,\mathit{nil})} * \langle\!\langle t'' \rangle\!\rangle^{(x,j),(m,\mathbf{a},\mathit{nil})} \end{array} \right\} \\
\qquad\qquad \texttt{while}\ x \neq \mathit{nil}\ \texttt{do} \\
\qquad\qquad\quad \left\{ \begin{array}{l} \exists m, j_1, d, x, t, t', t''.\, (t \otimes m[t'] \otimes t'' = t_1 \otimes \mathbf{b}[t_2]) \wedge x \neq \mathit{nil} \wedge m \Rrightarrow m * x \Rrightarrow x \\ \qquad \times \langle\!\langle t \rangle\!\rangle^{(i,j_1),(\mathit{nil},\mathbf{a},m)} * m \mapsto j_1, \mathbf{a}, d, x \\ \qquad\qquad * \langle\!\langle t' \rangle\!\rangle^{(d,-),(\mathit{nil},m,\mathit{nil})} * \langle\!\langle t'' \rangle\!\rangle^{(x,j),(m,\mathbf{a},\mathit{nil})} \end{array} \right\} \\
\qquad\qquad\quad \left\{ \begin{array}{l} \exists m, d, x, x', t, t', t''.\, (t \otimes x[t'] \otimes t'' = t_1 \otimes \mathbf{b}[t_2]) \wedge m \Rrightarrow m * x \Rrightarrow x \\ \qquad \times \langle\!\langle t \rangle\!\rangle^{(i,m),(\mathit{nil},\mathbf{a},x)} * x \mapsto m, \mathbf{a}, d, x' * \langle\!\langle t' \rangle\!\rangle^{(d,-),(\mathit{nil},x,\mathit{nil})} * \langle\!\langle t'' \rangle\!\rangle^{(x,j),(m,\mathbf{a},\mathit{nil})} \end{array} \right\} \\
\qquad\qquad\quad m := x; \\
\qquad\qquad\quad x := [m.\texttt{right}] \\
\qquad\qquad\quad \left\{ \begin{array}{l} \exists m, j_1, d, x, t, t', t''.\, (t \otimes m[t'] \otimes t'' = t_1 \otimes \mathbf{b}[t_2]) \wedge m \Rrightarrow m * x \Rrightarrow x \\ \qquad \times \langle\!\langle t \rangle\!\rangle^{(i,j_1),(\mathit{nil},\mathbf{a},m)} * m \mapsto j_1, \mathbf{a}, d, x \\ \qquad\qquad * \langle\!\langle t' \rangle\!\rangle^{(d,-),(\mathit{nil},m,\mathit{nil})} * \langle\!\langle t'' \rangle\!\rangle^{(x,j),(m,\mathbf{a},\mathit{nil})} \end{array} \right\} \\
\qquad\qquad \left\{ \begin{array}{l} \exists m, j_1, d, t, t', t''.\, (t \otimes m[t'] \otimes t'' = t_1 \otimes \mathbf{b}[t_2]) \wedge m \Rrightarrow m * x \Rrightarrow \mathit{nil} \\ \qquad \times \langle\!\langle t \rangle\!\rangle^{(i,j_1),(\mathit{nil},\mathbf{a},m)} * m \mapsto j_1, \mathbf{a}, d, \mathit{nil} \\ \qquad\qquad * \langle\!\langle t' \rangle\!\rangle^{(d,-),(\mathit{nil},m,\mathit{nil})} * \langle\!\langle t'' \rangle\!\rangle^{(\mathit{nil},j),(m,\mathbf{a},\mathit{nil})} \end{array} \right\} \\
\qquad\qquad \left\{ \begin{array}{l} \exists m, j_1, d, t, t'.\, (t \otimes m[t'] = t_1 \otimes \mathbf{b}[t_2]) \wedge m \Rrightarrow m * x \Rrightarrow \mathit{nil} \\ \qquad \times \langle\!\langle t \rangle\!\rangle^{(i,j_1),(\mathit{nil},\mathbf{a},m)} * m \mapsto j_1, \mathbf{a}, d, \mathit{nil} * \langle\!\langle t' \rangle\!\rangle^{(d,-),(\mathit{nil},m,\mathit{nil})} \end{array} \right\} \\
\qquad\qquad \left\{ \begin{array}{l} \exists j_1, d.\, m \Rrightarrow \mathbf{b} * x \Rrightarrow \mathit{nil} \\ \qquad \times \langle\!\langle t_1 \rangle\!\rangle^{(i,j_1),(\mathit{nil},\mathbf{a},\mathbf{b})} * \mathbf{b} \mapsto j_1, \mathbf{a}, d, \mathit{nil} * \langle\!\langle t_2 \rangle\!\rangle^{(d,-),(\mathit{nil},\mathbf{b},\mathit{nil})} \end{array} \right\} \\
\qquad\quad \left\{ m \Rrightarrow b * x \Rrightarrow \mathit{nil} \times \langle\!\langle t_1 \otimes \mathbf{b}[t_2] \rangle\!\rangle^{(i,j),(\mathit{nil},\mathbf{a},\mathit{nil})} \right\} \\
\qquad \left\{ \exists i, j.\, n \Rrightarrow \mathbf{a} * m \Rrightarrow \mathbf{b} * x \Rrightarrow \mathit{nil} \times \Cap^{F}_{(\mathbf{a},\mathbf{a}),(l,u,r)} * \mathbf{a} \mapsto l, u, i, r * \langle\!\langle t_1 \otimes \mathbf{b}[t_2] \rangle\!\rangle^{(i,j),(\mathit{nil},\mathbf{a},\mathit{nil})} \right\} \\
\quad \left\{ n \Rrightarrow \mathbf{a} * m \Rrightarrow \mathbf{b} \times \Cap^{F}_{(\mathbf{a},\mathbf{a}),(l,u,r)} * \langle\!\langle \mathbf{a}[t_1 \otimes \mathbf{b}[t_2]] \rangle\!\rangle^{(\mathbf{a},\mathbf{a}),(l,u,r)} \right\} \\
\}
\end{array}
$$

Fig. 5. Proof outline for `getLast` implementation (where node is present)

$$
\begin{array}{l}
m := \texttt{getLast}(n)\ \{ \\
\quad \left\{ n \Rrightarrow \mathbf{a} * m \Rrightarrow - \times \mathbb{m}^F_{(\mathbf{a},\mathbf{a}),(l,u,r)} * \langle\!\langle \mathbf{a}[\varnothing] \rangle\!\rangle^{(\mathbf{a},\mathbf{a}),(l,u,r)} \right\} \\
\quad \texttt{local } x \texttt{ in} \\
\qquad \left\{ n \Rrightarrow \mathbf{a} * m \Rrightarrow - * x \Rrightarrow - \times \mathbb{m}^F_{(\mathbf{a},\mathbf{a}),(l,u,r)} * \mathbf{a} \mapsto l, u, \mathit{nil}, r \right\} \\
\qquad m := [n.\texttt{down}]\,; \\
\qquad \left\{ n \Rrightarrow \mathbf{a} * m \Rrightarrow \mathit{nil} * x \Rrightarrow - \times \mathbb{m}^F_{(\mathbf{a},\mathbf{a}),(l,u,r)} * \mathbf{a} \mapsto l, u, \mathit{nil}, r \right\} \\
\qquad \texttt{if } m \neq \mathit{nil} \texttt{ then} \\
\qquad\quad \ldots \\
\qquad \left\{ n \Rrightarrow \mathbf{a} * m \Rrightarrow \mathit{nil} * x \Rrightarrow - \times \mathbb{m}^F_{(\mathbf{a},\mathbf{a}),(l,u,r)} * \mathbf{a} \mapsto l, u, \mathit{nil}, r \right\} \\
\quad \left\{ n \Rrightarrow \mathbf{a} * m \Rrightarrow \mathit{nil} \times \mathbb{m}^F_{(\mathbf{a},\mathbf{a}),(l,u,r)} * \langle\!\langle \mathbf{a}[\varnothing] \rangle\!\rangle^{(\mathbf{a},\mathbf{a}),(l,u,r)} \right\} \\
\}
\end{array}
$$

Fig. 6. Proof outline for `getLast` implementation (where node is absent)

5.5.1. *Soundness of* $\tau_2 : \mathbb{T} \to \mathbb{H} + \mathbb{L}$

Lemma 5.11 (τ_2 Application Preservation). For all $c \in \mathsf{C}_{\mathsf{Tree}}$, $t \in \mathsf{Tree}$ and $I \in \mathcal{I}$,

$$\langle\!\langle c \bullet t \rangle\!\rangle^I \equiv \exists I'.\, \langle\!\langle c \rangle\!\rangle^I_{I'} \bullet \langle\!\langle t \rangle\!\rangle^{I'}.$$

Proof. The proof is by induction on the structure of the context c, assuming some fixed $t \in \mathsf{Tree}$.

$c = -$ case:

$$
\begin{aligned}
\exists I'.\, \langle\!\langle - \rangle\!\rangle^I_{I'} \bullet \langle\!\langle t \rangle\!\rangle^{I'} &\equiv \exists I'.\, I = I' \wedge \langle\!\langle t \rangle\!\rangle^{I'} \\
&\equiv \langle\!\langle t \rangle\!\rangle^I \\
&\equiv \langle\!\langle - \bullet t \rangle\!\rangle^I.
\end{aligned}
$$

$c = \mathbf{a}[c']$ case:

$$
\begin{aligned}
&\exists I'.\, \langle\!\langle \mathbf{a}[c'] \rangle\!\rangle^{in,out}_{I'} \bullet \langle\!\langle t \rangle\!\rangle^{I'} \\
&\qquad \equiv \exists I'.\, \left(in = \mathbf{a} \wedge \exists a, l.\, \mathbf{a} \mapsto out, a * a \Mapsto [l] * \langle\!\langle c' \rangle\!\rangle^{l,\mathbf{a}}_{I'} \right) \bullet \langle\!\langle t \rangle\!\rangle^{I'} \\
&\qquad \equiv in = \mathbf{a} \wedge \exists a, l.\, \mathbf{a} \mapsto out, a * a \Mapsto [l] * \left(\exists I'.\, \langle\!\langle c' \rangle\!\rangle^{l,\mathbf{a}}_{I'} \bullet \langle\!\langle t \rangle\!\rangle^{I'} \right) \\
&\qquad \equiv in = \mathbf{a} \wedge \exists a, l.\, \mathbf{a} \mapsto out, a * a \Mapsto [l] * \langle\!\langle c' \bullet t \rangle\!\rangle^{l,\mathbf{a}} \\
&\qquad \equiv \langle\!\langle \mathbf{a}[c' \bullet t] \rangle\!\rangle^{in,out} \\
&\qquad \equiv \langle\!\langle \mathbf{a}[c'] \bullet t \rangle\!\rangle^{in,out}.
\end{aligned}
$$

$$E.\texttt{parent} \stackrel{\text{def}}{=} E$$

$$E.\texttt{children} \stackrel{\text{def}}{=} E + 1$$

$$n := \texttt{newNode}() \stackrel{\text{def}}{=} n := \texttt{alloc}(2)$$

$$\texttt{disposeNode}(E) \stackrel{\text{def}}{=} \texttt{dispose}(E, 2)$$

```
m := getUp(n) {
  local x in
    m := [n.parent];
    x := [m.parent];
    if x = nil then
      m := nil
}

m := getLast(n) {
  local x in
    x := [n.children];
    m := x.getTail()
}

newNodeAfter(n) {
  local x, y, z, w in
    x := [n.parent];
    z := [x.children];
    y := newNode();
    [y.parent] := x;
    z.insert(n, y);
    w := newList();
    [y.children] := w
}

m := getFirst(n) {
  local x in
    x := [n.children];
    m := x.getHead()
}

m := getRight(n) {
  local x, y in
    x := [n.parent];
    y := [x.children];
    m := y.getNext(n)
}

m := getLeft(n) {
  local x, y in
    x := [n.parent];
    y := [x.children];
    m := y.getPrev(n)
}

deleteTree(n) {
  local x, y, z in
    x := [n.parent];
    y := [x.children];
    y.remove(n);
    y := [n.children];
    z := y.getHead();
    while z ≠ nil do
      call deleteTree(z);
      z := y.getHead()  ;
    deleteList(y);
    disposeNode(n)
}
```

Fig. 7. Procedures for the list-based implementation of trees

$c = c' \otimes t'$ case:

$$\begin{aligned}
&\exists I'.\, \langle\!\langle c' \otimes t' \rangle\!\rangle_{I'}^{in,out} \bullet \langle\!\langle t \rangle\!\rangle^{I'} \\
&\qquad \equiv \exists I'.\, \left(\exists l_1, l_2.\, (in = l_1 \cdot l_2) \wedge \langle\!\langle c' \rangle\!\rangle_{I'}^{l_1,out} * \langle\!\langle t' \rangle\!\rangle^{l_2,out}\right) \bullet \langle\!\langle t \rangle\!\rangle^{I'} \\
&\qquad \equiv \exists l_1, l_2.\, (in = l_1 \cdot l_2) \wedge \left(\exists I'.\, \langle\!\langle c' \rangle\!\rangle_{I'}^{l_1,out} \bullet \langle\!\langle t \rangle\!\rangle^{I'}\right) * \langle\!\langle t' \rangle\!\rangle^{l_2,out} \\
&\qquad \equiv \exists l_1, l_2.\, (in = l_1 \cdot l_2) \wedge \langle\!\langle c' \bullet t \rangle\!\rangle^{l_1,out} * \langle\!\langle t' \rangle\!\rangle^{l_2,out} \\
&\qquad \equiv \langle\!\langle (c' \bullet t) \otimes t' \rangle\!\rangle^{in,out} \\
&\qquad \equiv \langle\!\langle (c' \otimes t') \bullet t \rangle\!\rangle^{in,out}.
\end{aligned}$$

The case where $c = t' \otimes c'$ follows the same pattern as the $c' \otimes t'$ case. □

Lemma 5.12 (τ_2 Crust Inclusion). For all $out, out' \in \mathcal{I}_{\text{out}}$, $F \in \mathcal{F}$ and $c \in \mathsf{C}_{\mathsf{Tree}}$, there exist $K \in \mathcal{P}(\mathsf{C}_{\mathbb{H}+\mathbb{L}})$ and $F' \in \mathcal{F}$ such that, for all $in \in \mathcal{I}_{\text{in}}$,

$$\left(\exists in'.\, \Subset_{in',out'}^{F} \circ \langle\!\langle c \rangle\!\rangle_{in,out}^{in',out'}\right) \equiv K \circ \Subset_{in,out}^{F'}.$$

Proof. The proof is by induction on the structure of the context c, assuming some fixed $out, out' \in \mathcal{I}_{\text{out}}$ and $F \in \mathcal{F}$.

$c = -$ case:
Choose $K = \text{emp}$ if $out' = out$ and $K = \text{False}$ otherwise; choose $F' = F$. If $out' \neq out$ then both sides of the equation are equivalent to False; otherwise, observe that

$$\begin{aligned}
\exists in'.\, \Subset_{in',out'}^{F} \circ \langle\!\langle - \rangle\!\rangle_{in,out}^{in',out'} &\equiv \Subset_{in,out}^{F} \\
&\equiv K \circ \Subset_{in,out}^{F}.
\end{aligned}$$

$c = \mathbf{a}[c']$ case:
By the inductive hypothesis, there exist K' and F' such that, for all in,

$$\exists l.\, \Subset_{l,\mathbf{a}}^{\varnothing,\varnothing,out'} \circ \langle\!\langle c' \rangle\!\rangle_{in,out}^{l,\mathbf{a}} \equiv K' \circ \Subset_{in,out}^{F'}.$$

Choose $K = \Subset_{\mathbf{a},out'}^{F} \circ K'$; choose F' as given above. Observe that

$$\begin{aligned}
&\exists in'.\, \Subset_{in',out'}^{F} \circ \langle\!\langle \mathbf{a}[c'] \rangle\!\rangle_{in,out}^{in',out'} \\
&\qquad \equiv \exists in'.\, \Subset_{in',out'}^{F} \circ \left(in' = \mathbf{a} \wedge \exists a, l.\, \mathbf{a} \mapsto out', a * a \Rrightarrow [l] * \langle\!\langle c' \rangle\!\rangle_{in,out}^{l,\mathbf{a}}\right) \\
&\qquad \equiv \Subset_{\mathbf{a},out'}^{F} \circ \exists l.\, \Subset_{l,\mathbf{a}}^{\varnothing,\varnothing,out'} \circ \langle\!\langle c' \rangle\!\rangle_{in,out}^{l,\mathbf{a}} \\
&\qquad \equiv \Subset_{\mathbf{a},out'}^{F} \circ K' \circ \Subset_{in,out}^{F'} \\
&\qquad \equiv K \circ \Subset_{in,out}^{F'}.
\end{aligned}$$

$c = c' \otimes t'$ case:
Observe that there is exactly one choice of $l_2 \in (\mathsf{Addr})^*$ such that $\langle\!\langle t' \rangle\!\rangle^{l_2,out'} \not\equiv \text{false}$. Let $\hat{l}_2$ be that choice. Observe also that there exists some K' such that

$$\langle\!\langle t' \rangle\!\rangle^{\hat{l}_2,out'} \equiv K' * \prod_{\mathbf{a} \in \hat{l}_2}^{*} \mathbf{a} \mapsto out'.$$

Let $(l_1', l_2', a') = F$. By the inductive hypothesis, there exist K'' and F' such that, for all in,

$$\exists l_1.\, \Cap{l_1', \hat{l}_2 \cdot l_2', a'}{l_1, out'} \circ \langle\!\langle c' \rangle\!\rangle^{l_1, out'}_{in, out} \equiv K'' \circ \Cap{F'}{in, out}.$$

Choose $K = K' \circ K''$; choose F' as given above. Observe that

$$\begin{aligned}
&\exists in'.\, \Cap{l_1', l_2', a'}{in', out'} \circ \langle\!\langle c' \otimes t' \rangle\!\rangle^{in', out'}_{in, out} \\
&\quad\equiv \exists in'. \left(\exists a.\, out' \mapsto a', a * a \Rightarrow [l_1' \cdot in' \cdot l_2'] * \prod^{*}_{\mathbf{a} \in l_1' \cdot l_2'} \mathbf{a} \mapsto out' \right) \\
&\qquad\qquad \circ \exists l_1.\, in' = l_1 \cdot \hat{l}_2 \wedge \langle\!\langle c' \rangle\!\rangle^{l_1, out'}_{in, out} * \langle\!\langle t' \rangle\!\rangle^{\hat{l}_2, out'} \\
&\quad\equiv \exists l_1. \left(\exists a.\, out' \mapsto a', a * a \Rightarrow \left[l_1' \cdot l_1 \cdot \hat{l}_2 \cdot l_2' \right] * \prod^{*}_{\mathbf{a} \in l_1' \cdot l_2'} \mathbf{a} \mapsto out' \right) \\
&\qquad\qquad \circ \langle\!\langle c' \rangle\!\rangle^{l_1, out'}_{in, out} * K' * \prod^{*}_{\mathbf{a} \in \hat{l}_2} \mathbf{a} \mapsto out' \\
&\quad\equiv K' \circ \exists l_1.\, \Cap{l_1', \hat{l}_2 \cdot l_2', a'}{l_1, out'} \circ \langle\!\langle c' \rangle\!\rangle^{l_1, out'}_{in, out} \\
&\quad\equiv K' \circ K'' \circ \Cap{F'}{in, out} \\
&\quad\equiv K \circ \Cap{F'}{in, out}.
\end{aligned}$$

The case where $c = t' \otimes c'$ follows the same pattern as the $c' \otimes t'$ case. □

Lemma 5.13 (τ_2 Axiom Correctness). For all $\varphi \in \mathsf{Cmd}_{\mathbb{L}}$, $(P, Q) \in \mathrm{Ax}\,[\![\varphi]\!]_{\mathbb{L}}$, $out \in \mathcal{I}_{\mathrm{out}}$ and $F \in \mathcal{F}$,

$$\vdash_{\mathbb{H}+\mathbb{L}} \left\{ (\!| P |\!)^{out, F} \right\} \, [\![\varphi]\!] \, \left\{ (\!| Q |\!)^{out, F} \right\}.$$

We omit the full proof details here, which are due to Wheelhouse and can be found in (Dinsdale-Young et al., 2010b). Fig. 8 shows one of the simpler proof cases involved in establishing axiom correctness, namely for the `getUp` axiom.

This completes the proof of Theorem 5.10.

5.6. *Module Translation:* $\tau_3 : \mathbb{H} + \mathbb{H} \to \mathbb{H}$

Another example of a locality-preserving translation is the natural implementation of a pair of heap modules $\mathbb{H} + \mathbb{H}$ with a single heap module $\mathbb{H}$ that simply treats the two heaps as (disjoint) portions of the same heap. This example is an important one, as it does not result in a surjective abstraction relation (different abstract heap pairs can map to the same concrete heap), but is still a sound locality preserving translation.

Definition 5.7 ($\tau_3 : \mathbb{H}+\mathbb{H} \to \mathbb{H}$). The pre-locality-preserving translation $\tau_3 : \mathbb{H}+\mathbb{H} \to \mathbb{H}$ is constructed as follows:

$$
\begin{array}{l}
m := \texttt{getUp}(n)\ \{ \\
\quad \left\{\ m \Rrightarrow - * n \Rrightarrow \mathbf{b} \times \exists in.\, \Cap^F_{in,out} \bullet \langle\!\langle \mathbf{a}[t_1 \otimes \mathbf{b}[t_2] \otimes t_3] \rangle\!\rangle^{in,out}\ \right\} \\
\quad \left\{\begin{array}{l} m \Rrightarrow - * n \Rrightarrow \mathbf{b} \times \Cap^F_{\mathbf{a},out} \bullet \exists a, l_1, l_3.\, \mathbf{a} \mapsto out, a * a \Mapsto [l_1 \cdot \mathbf{b} \cdot l_3] \\ \qquad * \langle\!\langle t_1 \rangle\!\rangle^{l_1,\mathbf{a}} * \exists a', l.\, \mathbf{b} \mapsto \mathbf{a}, a' * a' \Mapsto [l_2] * \langle\!\langle t_2 \rangle\!\rangle^{l_2,\mathbf{b}} * \langle\!\langle t_3 \rangle\!\rangle^{l_3,\mathbf{a}} \end{array}\right\} \\
\qquad \left\{\ m \Rrightarrow - * n \Rrightarrow \mathbf{b} \times \mathbf{a} \mapsto out * \mathbf{b} \mapsto \mathbf{a}\ \right\} \\
\qquad \texttt{local } x \texttt{ in} \\
\qquad\quad \left\{\ m \Rrightarrow - * n \Rrightarrow \mathbf{b} * x \Rrightarrow - \times \mathbf{a} \mapsto out * \mathbf{b} \mapsto \mathbf{a}\ \right\} \\
\qquad\quad m := [n.\texttt{parent}]\,; \\
\qquad\quad \left\{\ m \Rrightarrow \mathbf{a} * n \Rrightarrow \mathbf{b} * x \Rrightarrow - \times \mathbf{a} \mapsto out * \mathbf{b} \mapsto \mathbf{a}\ \right\} \\
\qquad\quad x := [m.\texttt{parent}]\,; \\
\qquad\quad \left\{\ m \Rrightarrow \mathbf{a} * n \Rrightarrow \mathbf{b} * x \Rrightarrow out \times \mathbf{a} \mapsto out * \mathbf{b} \mapsto \mathbf{a}\ \right\} \\
\qquad\quad \texttt{if } x = \mathit{nil} \texttt{ then} \\
\qquad\qquad m := \mathit{nil} \\
\qquad\quad \left\{\ m \Rrightarrow \mathbf{a} * n \Rrightarrow \mathbf{b} * x \Rrightarrow out \times \mathbf{a} \mapsto out * \mathbf{b} \mapsto \mathbf{a}\ \right\} \\
\qquad \left\{\ m \Rrightarrow \mathbf{a} * n \Rrightarrow \mathbf{b} \times \mathbf{a} \mapsto out * \mathbf{b} \mapsto \mathbf{a}\ \right\} \\
\quad \left\{\begin{array}{l} m \Rrightarrow \mathbf{a} * n \Rrightarrow \mathbf{b} \times \Cap^F_{\mathbf{a},out} \bullet \exists a, l_1, l_3.\, \mathbf{a} \mapsto out, a * a \Mapsto [l_1 \cdot \mathbf{b} \cdot l_3] \\ \qquad * \langle\!\langle t_1 \rangle\!\rangle^{l_1,\mathbf{a}} * \exists a', l.\, \mathbf{b} \mapsto \mathbf{a}, a' * a' \Mapsto [l_2] * \langle\!\langle t_2 \rangle\!\rangle^{l_2,\mathbf{b}} * \langle\!\langle t_3 \rangle\!\rangle^{l_3,\mathbf{a}} \end{array}\right\} \\
\quad \left\{\ m \Rrightarrow \mathbf{a} * n \Rrightarrow \mathbf{b} \times \exists in.\, \Cap^F_{in,out} \bullet \langle\!\langle \mathbf{a}[t_1 \otimes \mathbf{b}[t_2] \otimes t_3] \rangle\!\rangle^{in,out}\ \right\} \\
\}
\end{array}
$$

Fig. 8. Proof outline for `getUp` implementation

— the interface sets are both unit sets,[¶] *i.e.* $\mathcal{I}_{\text{in}} = \{1\} = \mathcal{I}_{\text{out}}$, and hence they can be ignored;
— the representation function (which is the same for both data and contexts) is defined as

$$\langle\!\langle (h_1, h_2) \rangle\!\rangle \quad \stackrel{\text{def}}{=} \quad \{h_1\} * \{h_2\}\,;$$

— the crust parameter set is also the unit set and the crust predicate is defined as $\Cap = \text{emp}$; and
— the implementation function is given by replacing the commands for both heaps with their detagged versions, for example

$$[\![n := \texttt{alloc}_1(E)]\!] \quad = \quad n := \texttt{alloc}(E) \quad = \quad [\![n := \texttt{alloc}_2(E)]\!]\,.$$

Theorem 5.14 (Soundness of τ_3). The pre-locality-preserving translation τ_3 is a locality-preserving translation.

This theorem is trivial: application preservation holds by properties of $*$; crust inclusion holds since the crust is simply emp; and axiom correctness holds because the axioms of $\mathbb{H}+\mathbb{H}$ are almost directly translated to those of $\mathbb{H}$, modulo a frame. Notice, however, that

[¶] It makes no difference which set, as long as it only has one element.

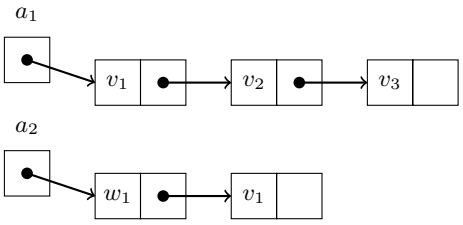

Fig. 9. Representation of the list store $a_1 \Mapsto [v_1 \cdot v_2 \cdot v_3] * a_2 \Mapsto [w_1 \cdot v_1]$ as singly linked lists in the heap

this translation does not satisfy the first of the properties for including the conjunction rule (§5.3), since $\langle\!\langle (1 \mapsto 0, \mathrm{emp}) \rangle\!\rangle = \{1 \mapsto 0\} = \langle\!\langle (\mathrm{emp}, 1 \mapsto 0) \rangle\!\rangle$.

If a command like `allocEither` from §2 were added to $\mathbb{H} + \mathbb{H}$, then without Conj it could be soundly implemented in $\mathbb{H}$ by allocating a single cell and returning its address. With Conj, however, this does not work; the implementation must diverge.

6. Locality-Breaking Translations

There is not always a close correspondence between the locality exhibited by a high-level module and the locality of the low-level module on which it is implemented. In this section, we reduce the burden of proof for a sound module translation in such cases by introducing *locality-breaking translations*. In §6.1, we establish that locality-breaking translations give sound module translations, and in §6.3 we give a locality-breaking translation $\tau_4 : \mathbb{L} \to \mathbb{H}$.

Our motivating example is an implementation of the list module (as defined in §3.3) that represents each abstract list with a singly-linked list in the heap. An example of a list store represented in this way is depicted in Fig. 9. Consider the operation of removing the value v_3 from the list at address a_1. At the abstract level, the resource required by the operation is simply $a_1 \Mapsto v_3$, but in the implementation the list at a_1 must be traversed all the way to the node labelled v_3, in this case this is the entire list.

To apply the locality-preserving translation approach, an arbitrary amount of additional data would have to be included in the crust. This seems to defeat the purpose of the locality-preserving approach in the first place, so perhaps a more direct approach is called for.

In order to prove the soundness of a module translation, it is necessary to demonstrate a transformation from high-level proofs about programs that use an abstract module, to low-level proofs of those programs using the concrete module implementation. Inductively transforming proofs is, intuitively, a good strategy because the high-level rules are matched by the low-level rules. Since the definition of predicate translations preserves disjunctions and entailments, and also the variable scope, the majority of the proof rules can be inductively transformed to their low-level counterparts directly. The two exceptions to this are Frame and Axiom. For locality-preserving translations, Frame was dealt with by the fact that a locality-preserving translation preserves context application,

and Axiom was dealt with by the fact that the implementations of the basic commands satisfy the axioms under such a translation.

When it is not appropriate to consider a translation that preserves locality, it would be useful if it were only necessary to consider proofs which do not use the frame rule, or, at least, only use it in a limited fashion. Intuitively, of course this should be possible. The purpose of the frame rule is to factor out parts of the state that do not play a role (and hence, do not change) in part of the program under consideration. It is common folklore that it should be possible to transform a proof to one in which the frame rule is only applied to basic statements (*i.e.* basic commands and assignment) by factoring in the state earlier in the proof (*i.e.* at the leaves of the proof). This intuition is formalised in the following lemma.

Lemma 6.1 (Frame-Free Derivations). Let $\mathbb{A}$ be an abstract module. If there is a proof derivation of $\vdash_{\mathbb{A}} \{P\}\ \mathbb{C}\ \{Q\}$ then there is also a derivation that only uses the frame rule in the following ways:

$$\dfrac{\dfrac{}{\Gamma \vdash_{\mathbb{A}} \{P'\}\ \mathbb{C}'\ \{Q'\}}\ (\dagger)}{\Gamma \vdash_{\mathbb{A}} \{K \bullet P'\}\ \mathbb{C}'\ \{K \bullet Q'\}}\ \textsc{Frame} \tag{2}$$

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash_{\mathbb{A}} \{P'\}\ \mathbb{C}'\ \{Q'\}}}{\Gamma \vdash_{\mathbb{A}} \{(K_{\mathsf{Scope}} \times \mathbf{I}_{\mathbb{A}}) \bullet P'\}\ \mathbb{C}'\ \{(K_{\mathsf{Scope}} \times \mathbf{I}_{\mathbb{A}}) \bullet Q'\}}\ \textsc{Frame} \tag{3}$$

where (†) is either Axiom or Assgn.

Consider a translation $\tau : \mathbb{A} \to \mathbb{B}$. Lemma 6.1 implies that it is only necessary to provide proofs of $\vdash_{\mathbb{B}} \{[\![P]\!]_\tau\}\ [\![\mathbb{C}]\!]_\tau\ \{[\![Q]\!]_\tau\}$ when there is a proof of $\vdash_{\mathbb{A}} \{P\}\ \mathbb{C}\ \{Q\}$ having the prescribed form. Provided that there are proofs that the implementation of each command in $\mathsf{Cmd}_{\mathbb{A}}$ satisfies the translation of its axioms under every possible frame, the proof in $\mathbb{A}$ can be transformed to a proof in $\mathbb{B}$ by straightforward induction. In fact, we only need to consider *singleton* frames, as we can treat an arbitrary frame as the disjunction of singletons and apply the Disj rule. We can further reduce our considerations to those singleton frames with no variable scope component, since the variable scope component can be added by the Frame rule at the low-level. These conditions are formalised in the definition of a locality-breaking translation.

Definition 6.1 (Locality-Breaking Translation). A *locality-breaking translation* $\tau : \mathbb{A} \to \mathbb{B}$ is a module translation having the property that, for all $c \in \mathsf{C}_{\mathbb{A}}$, $\varphi \in \mathsf{Cmd}_{\mathbb{A}}$, and $(P, Q) \in \mathrm{Ax}\,[\![\varphi]\!]_{\mathbb{A}}$ there is a derivation of

$$\vdash_{\mathbb{B}} \{[\![\{(\varnothing, c)\} \bullet P]\!]_\tau\}\ [\![\varphi]\!]_\tau\ \{[\![\{(\varnothing, c)\} \bullet Q]\!]_\tau\}\,.$$

Theorem 6.2 (Soundness of Locality-Breaking Translations). A locality-breaking translation is a sound module translation.

A locality-breaking translation transforms proofs that use locality (in the form of the Frame rule) at the abstract level into proofs that do not. To do so, we must effectively

prove directly that the abstract FRAME rule is sound with respect to the implementation of each module operation. Hence we say that such a module translation provides a *fiction of locality*.

Since a locality-breaking translation transforms proofs that use locality (in the form of the frame rule) into proofs that do not, we

6.1. *Proof of Soundness of Locality-Breaking Translations*

Proof of Lemma 6.1 The result is a special case of the more general result, that if there is a derivation of $\Gamma \vdash_{\mathbb{A}} \{P\}\ \mathbb{C}\ \{Q\}$ then there is a derivation of $F(\Gamma) \vdash_{\mathbb{A}} \{P\}\ \mathbb{C}\ \{Q\}$ with the required property, where

$$F(\Gamma) = \{\mathtt{f} : K \bullet \mathsf{P} \rightarrowtail K \bullet \mathsf{Q} \mid K \in \mathcal{P}(\mathsf{C}_{\mathbb{A}}) \text{ and } (\mathtt{f} : \mathsf{P} \rightarrowtail \mathsf{Q}) \in \Gamma\}\,.$$

Note that $\Gamma \subseteq F(\Gamma) = F(F(\Gamma))$. Since procedure specifications are only relevant to the PDEF and PCALL rules, we will omit them when considering other rules.

The proof of the generalised statement is by induction on the depth of the derivation. If the last rule applied in the derivation is anything other than FRAME or PDEF then it is simple to transform the derivation: simply apply the induction hypothesis to transform all of the premises and then apply the last rule using $F(\gamma)$ in place of γ.

Consider the case where the last rule of the derivation is FRAME:

$$\dfrac{\dfrac{\vdots}{\{P'\}\ \mathbb{C}\ \{Q'\}}\ (\ddagger)}{\{K \bullet P'\}\ \mathbb{C}\ \{K \bullet Q'\}}\ \textsc{Frame}$$

By applying the disjunction rule, this can be reduced to the case of singleton frames $\{c\}$, transforming the derivation as follows:

$$\dfrac{\text{for all } c \in K, \quad \dfrac{\dfrac{\vdots}{\{P'\}\ \mathbb{C}\ \{Q'\}}\ (\ddagger)}{\{\{c\} \bullet P'\}\ \mathbb{C}\ \{\{c\} \bullet Q'\}}\ \textsc{Frame}}{\{K \bullet P'\}\ \mathbb{C}\ \{K \bullet Q'\}}\ \textsc{Disj}$$

Now consider cases for $(\ddagger)$, the last rule applied before FRAME.

If the rule is CONS then, since $P \subseteq P'$ implies that $\{c\} \bullet P \subseteq \{c\} \bullet P$, the application of the FRAME can be moved earlier in the derivation, transforming it as follows:

$$\dfrac{\begin{array}{ll}\{c\} \bullet P' \subseteq \{c\} \bullet P'' & \dfrac{\{P''\}\ \mathbb{C}\ \{Q''\}}{\{\{c\} \bullet P''\}\ \mathbb{C}\ \{\{c\} \bullet Q''\}}\ \textsc{Frame} \\ \{c\} \bullet Q' \subseteq \{c\} \bullet Q'' & \end{array}}{\{\{c\} \bullet P'\}\ \mathbb{C}\ \{\{c\} \bullet Q'\}}\ \textsc{Cons}$$

The application of the frame rule can then by removed by the inductive hypothesis.

If the rule is DISJ then, since $\bullet$ right-distributes over $\vee$, the derivation can be trans-

formed as follows:

$$\dfrac{\text{for all } i \in I, \quad \dfrac{\dfrac{\vdots}{\{P_i\}\ \mathbb{C}\ \{Q_i\}}}{\{\{c\} \bullet P_i\}\ \mathbb{C}\ \{\{c\} \bullet Q_i\}}\ \textsc{Frame}}{\left\{\{c\} \bullet \bigvee_{i\in I} P_i\right\}\ \mathbb{C}\ \left\{\{c\} \bullet \bigvee_{i\in I} Q_i\right\}}\ \textsc{Disj}$$

If the rule is Local then it is possible that the frame c includes a program variable with the same name as one that is scoped by the `local` block. This means that the frame cannot in general be pushed into the local block. However, the frame can be split into scope and store components, that is, for some $\rho \in \mathsf{Scope}$ and $c_{\mathbb{A}} \in \mathsf{C}_{\mathbb{A}}$, $c = (\rho, c_{\mathbb{A}})$ and so $\{c\} = (\{\rho\} \times \mathbf{I}_{\mathbb{A}}) \circ \{(\varnothing, c_{\mathbb{A}})\}$. Hence the derivation can be transformed as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\{(\boldsymbol{x} \Rrightarrow - \times \mathbf{I}_{\mathbb{A}}) \bullet P'\}\ \mathbb{C}'\ \{(\boldsymbol{x} \Rrightarrow - \times \mathbf{I}_{\mathbb{A}}) \bullet Q'\}}\ \textsc{Frame}}{\{(\boldsymbol{x} \Rrightarrow - \times \{c_{\mathbb{A}}\}) \bullet P'\}\ \mathbb{C}'\ \{(\boldsymbol{x} \Rrightarrow - \times \{c_{\mathbb{A}}\}) \bullet Q'\}}\ \textsc{Local}}{\{\{(\varnothing, c_{\mathbb{A}})\} \bullet P'\}\ \texttt{local}\ \boldsymbol{x}\ \texttt{in}\ \mathbb{C}'\ \{\{(\varnothing, c_{\mathbb{A}})\} \bullet Q'\}}\ \textsc{Frame}}{\{\{c\} \bullet P'\}\ \texttt{local}\ \boldsymbol{x}\ \texttt{in}\ \mathbb{C}'\ \{\{c\} \bullet Q'\}}$$

The side condition for the Local rule, that $(\{(\varnothing, c_{\mathbb{A}})\} \bullet P') \wedge \mathit{vsafe}(\boldsymbol{x}) \equiv \emptyset$, follows from the original side-condition that $P' \wedge \mathit{vsafe}(\boldsymbol{x}) \equiv \emptyset$. The applications of the Frame rule are now either of the form of (3) or can be removed by the inductive hypothesis.

If the rule is PCall then it is again necessary to split the framing context into its components, that is, some $\rho \in \mathsf{Scope}$ and $c_{\mathbb{A}} \in \mathsf{C}_{\mathbb{A}}$ with $c = (\rho, c_{\mathbb{A}})$. The PCall rule uses some $\mathtt{f} : \mathsf{P} \rightarrowtail \mathsf{Q} \in \Gamma$. By definition, $\mathtt{f} : \{c_{\mathbb{A}}\} \bullet \mathsf{P} \rightarrowtail \{c_{\mathbb{A}}\} \bullet \mathsf{Q} \in F(\Gamma)$. Hence, the derivation can be transformed as follows:

$$\dfrac{\dfrac{\{(\overrightarrow{\boldsymbol{r}} \Rrightarrow \overrightarrow{w} * \rho')\} \times \mathsf{D}_{\mathbb{A}} \subseteq \mathit{vsafe}(\overrightarrow{E})}{F(\Gamma) \vdash_{\mathbb{A}} \left\{(\overrightarrow{\boldsymbol{r}} \Rrightarrow \overrightarrow{v} * \rho') \times \left(\{(c_{\mathbb{A}})\} \bullet \mathsf{P}\left(\mathcal{E}\left[\!\left[\overrightarrow{E}\right]\!\right](\overrightarrow{\boldsymbol{r}} \Rrightarrow \overrightarrow{v} * \rho')\right)\right)\right\}\ \texttt{call}\ \overrightarrow{\boldsymbol{r}} := \mathtt{f}(\overrightarrow{E})}\ \textsc{PCall}}{F(\Gamma) \vdash_{\mathbb{A}} \left\{\{(\rho, c_{\mathbb{A}})\} \bullet \left((\overrightarrow{\boldsymbol{r}} \Rrightarrow \overrightarrow{v} * \rho') \times \mathsf{P}\left(\mathcal{E}\left[\!\left[\overrightarrow{E}\right]\!\right](\overrightarrow{\boldsymbol{r}} \Rrightarrow \overrightarrow{v} * \rho')\right)\right)\right\}\ \texttt{call}\ \overrightarrow{\boldsymbol{r}} := \mathtt{f}(\overrightarrow{E})}\ \textsc{Frame}$$

$$\{\{(\rho, c_{\mathbb{A}})\} \bullet (\exists \overrightarrow{w}.\, \{(\overrightarrow{\boldsymbol{r}} \Rrightarrow \overrightarrow{w} * \rho')\} \times \mathsf{Q}(\overrightarrow{w}))\}$$

The application of the frame rule is now of the form of (3), with the frame being $\{\rho\} \times \mathbf{I}_{\mathbb{A}}$.

The remaining cases for the penultimate rule are straightforward.

Consider the case when PDEF is the last rule applied:

$$\dfrac{\text{for all } (\mathtt{f}_i : \mathsf{P} \rightarrowtail \mathsf{Q}) \in \Gamma, \quad \dfrac{\vdots}{\Gamma', \Gamma \vdash_{\mathbb{A}} \begin{array}{c}\{\exists \overrightarrow{v}.\, \overrightarrow{\boldsymbol{x}_i} \Rrightarrow \overrightarrow{v} * \overrightarrow{\boldsymbol{r}_i} \Rrightarrow - \times \mathsf{P}(\overrightarrow{v})\} \\ \mathbb{C}_i \\ \{\exists \overrightarrow{w}.\, \overrightarrow{\boldsymbol{x}_i} \Rrightarrow - * \overrightarrow{\boldsymbol{r}_i} \Rrightarrow \overrightarrow{w} \times \mathsf{Q}(\overrightarrow{w})\}\end{array}}{(\star)} \qquad (\star) \quad \dfrac{\vdots}{\Gamma', \Gamma \vdash_{\mathbb{A}} \{P\}\ \mathbb{C}'\ \{Q\}}}{\Gamma' \vdash_{\mathbb{A}} \{P\}\ \mathtt{procs}\ \overrightarrow{\boldsymbol{r}_1} := \mathtt{f}_1(\overrightarrow{\boldsymbol{x}_1})\ \{\mathbb{C}_1\}, \ldots, \overrightarrow{\boldsymbol{r}_k} := \mathtt{f}_k(\overrightarrow{\boldsymbol{x}_k})\ \{\mathbb{C}_k\}\ \mathtt{in}\ \mathbb{C}'\ \{Q\}}\ \textsc{PDef}$$

The derivations for the function bodies can be extended by applying the frame rule to give:

$$\dfrac{\text{for all } K \in \mathcal{P}(\mathsf{C}_{\mathbb{A}}), \\ (\mathtt{f}_i : \mathsf{P} \rightarrowtail \mathsf{Q}) \in \Gamma, \quad \dfrac{\vdots}{\Gamma', \Gamma \vdash_{\mathbb{A}} \begin{array}{c}\{\exists \overrightarrow{v}.\, \overrightarrow{\boldsymbol{x}_i} \Rrightarrow \overrightarrow{v} * \overrightarrow{\boldsymbol{r}_i} \Rrightarrow - \times \mathsf{P}(\overrightarrow{v})\} \\ \mathbb{C}_i \\ \{\exists \overrightarrow{w}.\, \overrightarrow{\boldsymbol{x}_i} \Rrightarrow - * \overrightarrow{\boldsymbol{r}_i} \Rrightarrow \overrightarrow{w} \times \mathsf{Q}(\overrightarrow{w})\}\end{array}}{\Gamma', \Gamma \vdash_{\mathbb{A}} \begin{array}{c}\{\exists \overrightarrow{v}.\, \overrightarrow{\boldsymbol{x}_i} \Rrightarrow \overrightarrow{v} * \overrightarrow{\boldsymbol{r}_i} \Rrightarrow - \times (K \bullet \mathsf{P}(\overrightarrow{v}))\} \\ \mathbb{C}_i \\ \{\exists \overrightarrow{w}.\, \overrightarrow{\boldsymbol{x}_i} \Rrightarrow - * \overrightarrow{\boldsymbol{r}_i} \Rrightarrow \overrightarrow{w} \times (K \bullet \mathsf{Q}(\overrightarrow{w}))\}\end{array}}\ \textsc{Frame}$$

These derivations and the derivation of the premise $\Gamma', \Gamma \vdash_{\mathbb{A}} \{P\}\ \mathbb{C}'\ \{Q\}$ can be transformed by the inductive hypothesis so that they only use the frame rule in the required manner and use the procedure environment $F(\Gamma', \Gamma) = F(\Gamma'), F(\Gamma)$. These derivations can then be recombined to give the required derivation as follows:

$$\dfrac{\text{for all } (\mathtt{f}_i : \mathsf{P} \rightarrowtail \mathsf{Q}) \in F(\Gamma), \quad \dfrac{\vdots}{F(\Gamma', \Gamma) \vdash_{\mathbb{A}} \begin{array}{c}\{\exists \overrightarrow{v}.\, \overrightarrow{\boldsymbol{x}_i} \Rrightarrow \overrightarrow{v} * \overrightarrow{\boldsymbol{r}_i} \Rrightarrow - \times \mathsf{P}(\overrightarrow{v})\} \\ \mathbb{C}_i \\ \{\exists \overrightarrow{w}.\, \overrightarrow{\boldsymbol{x}_i} \Rrightarrow - * \overrightarrow{\boldsymbol{r}_i} \Rrightarrow \overrightarrow{w} \times \mathsf{Q}(\overrightarrow{w})\}\end{array}}{(\star)} \qquad (\star) \quad \dfrac{\vdots}{F(\Gamma', \Gamma) \vdash_{\mathbb{A}} \{P\}\ \mathbb{C}'\ \{Q\}}}{\begin{array}{c}\{P\} \\ F(\Gamma') \vdash_{\mathbb{A}} \mathtt{procs}\ \overrightarrow{\boldsymbol{r}_1} := \mathtt{f}_1(\overrightarrow{\boldsymbol{x}_1})\ \{\mathbb{C}_1\}, \ldots, \overrightarrow{\boldsymbol{r}_k} := \mathtt{f}_k(\overrightarrow{\boldsymbol{x}_k})\ \{\mathbb{C}_k\}\ \mathtt{in}\ \mathbb{C}' \\ \{Q\}\end{array}}\ \textsc{PDef}$$

The two further conditions on the PDEF rule, not included above, hold for the transformed derivation because F preserves the names of the functions in procedure specifications. □

Let $\tau : \mathbb{A} \to \mathbb{B}$ be a locality-breaking translation. To show that τ is a sound module translation, it is necessary to establish that whenever there is a derivation of $\vdash_{\mathbb{A}} \{P\}\ \mathbb{C}\ \{Q\}$ there is a derivation of $\vdash_{\mathbb{B}} \{[\![P]\!]_\tau\}\ [\![\mathbb{C}]\!]_\tau\ \{[\![Q]\!]_\tau\}$. First, transform the high-level derivation using Lemma 6.1 into a frame-free derivation. Now transform this deriva-

tion to the required low-level derivation by replacing each subderivation of the form

$$\dfrac{\dfrac{}{\Gamma \vdash_{\mathbb{A}} \{P'\}\ \varphi\ \{Q'\}}\ \textsc{Axiom}}{\Gamma \vdash_{\mathbb{A}} \{K \bullet P'\}\ \varphi\ \{K \bullet Q'\}}\ \textsc{Frame}$$

with the derivation

$$\dfrac{\text{for all } (\rho, c_{\mathbb{A}}) \in K, \quad \dfrac{\dfrac{(\star)}{\vdash_{\mathbb{B}} \{[\![\{(\varnothing, c_{\mathbb{A}}) \bullet P'\}]\!]\}\ [\![\varphi]\!]\ \{[\![\{(\varnothing, c_{\mathbb{A}}) \bullet Q'\}]\!]\}}\ \textsc{Frame}}{\vdash_{\mathbb{B}} \{[\![\{(\rho, c_{\mathbb{A}}) \bullet P'\}]\!]\}\ [\![\varphi]\!]\ \{[\![\{(\rho, c_{\mathbb{A}}) \bullet Q'\}]\!]\}}\ \textsc{Frame}}{\vdash_{\mathbb{B}} \{[\![K \bullet P']\!]\}\ [\![\varphi]\!]\ \{[\![K \bullet Q']\!]\}}\ \textsc{Disj}$$

where $(\star)$ stands for the framed derivation provided by the locality-breaking translation, and replacing all other rules with their low-level equivalents.

This completes the proof of Theorem 6.2.

6.2. *Including the Conjunction Rule*

If we wish to add the conjunction rule to the locality-breaking theory, we can be add a case to the proof of Lemma 6.1 to deal with pushing Frame over Conj, in a similar fashion to Disj, provided that the context algebra $\mathcal{A}_{\mathbb{A}}$ is left-cancellative.

Definition 6.2 (Left-cancellativity). A context algebra $\mathcal{A} = (\mathsf{C}, \mathsf{D}, \circ, \bullet, \mathbf{I}, \mathbf{0})$ is left-cancellative if, for all $c \in \mathsf{C}$, $d_1, d_2, d_3 \in \mathsf{D}$, $c \bullet d_1 = c \bullet d_2 = d_3$ implies $d_1 = d_2$.

Left-cancellativity ensures that $\{c\} \bullet \bigwedge_{i \in I} P_i \equiv \bigwedge_{i \in I} \{c\} \bullet P_i$. It is also necessary for the predicate translation $[\![(\cdot)]\!]$ to distribute over conjunction; this is equivalent to the condition that the abstraction relation α is functional (that is, it defines a partial function from concrete states to abstract states).

6.3. *Module Translation:* $\tau_4 : \mathbb{L} \to \mathbb{H}$

We return to the example of an implementation of the list-store module $\mathbb{L}$ with singly-linked lists in the heap module $\mathbb{H}$ to illustrate a locality-breaking translation.

Definition 6.3 ($\tau_4 : \mathbb{L} \to \mathbb{H}$). The module translation $\tau_4 : \mathbb{L} \to \mathbb{H}$ is constructed as follows:

— the abstraction relation $\alpha_{\tau_4} \subseteq \mathsf{Heap} \times \mathsf{LStore}$ is defined by

$$h\ \alpha_{\tau_4}\ ls \iff h \in (\!|ls|\!)$$

where $(\!|(\cdot)|\!) : \mathsf{LStore} \to \mathcal{P}(\mathsf{Heap})$ is defined inductively as follows:

$$\begin{array}{rcl} (\!|\mathrm{emp}|\!) & \stackrel{\text{def}}{=} & \mathrm{emp} \\ (\!|a \mapsto l * ls|\!) & \stackrel{\text{def}}{=} & \mathrm{false} \\ (\!|a \mapsto [l] * ls|\!) & \stackrel{\text{def}}{=} & \exists x.\, a \mapsto x * \langle\!\langle l \rangle\!\rangle^{(x, \mathbf{nil})} * (\!|ls|\!) \end{array}$$

where

$$\begin{array}{rcl} \langle\!\langle \varnothing \rangle\!\rangle^{(x,y)} & \stackrel{\text{def}}{=} & (x = y) \wedge \text{emp} \\ \langle\!\langle v \rangle\!\rangle^{(x,y)} & \stackrel{\text{def}}{=} & x \mapsto v, y \\ \langle\!\langle l_1 \cdot l_2 \rangle\!\rangle^{(x,y)} & \stackrel{\text{def}}{=} & \exists z.\, \langle\!\langle l_1 \rangle\!\rangle^{(x,z)} * \langle\!\langle l_2 \rangle\!\rangle^{(z,y)}; \text{ and} \end{array}$$

— the substitutive implementation function is given by replacing each list-module command with a call to the correspondingly-named procedure given in Figs. 10 and 11.

Note that the abstraction relation is not surjective, since incomplete lists do not have heap representations (they are mapped to false). The intuition behind this approach is that incomplete lists are purely a useful means to the ultimate end of reasoning about complete lists. Typically, clients of the list module will work with complete lists (only complete lists may be created or deleted, for a start) and so restricting their specifications to only describe stores of complete lists should be no significant problem. Of course, it is perfectly acceptable to use assertions and specifications that refer to incomplete lists within client proofs; the transformation of this proof to a low-level proof will complete all of these lists by making use of Lemma 6.1.

The fact that incomplete lists do not have representations can be seen as an advantage from the point of view of establishing that τ_4 is a locality-breaking translation. This is because it is only necessary to prove that the framed axioms hold under the translation for frames that complete all of the lists in the precondition. In all other cases, the precondition is false, and so the triple holds trivially.

Theorem 6.3 (Soundness of τ_4). The module translation τ_4 is a locality-breaking translation.

Again, we omit the full details of this proof, which can be found in (Dinsdale-Young et al., 2010b), but give a particular case: `getNext`. Recall the axiom for `getNext`:

$$\begin{array}{l} \text{Ax}\,[\![\boldsymbol{x} := E_1.\texttt{getNext}(E_2)]\!]_{\mathbb{L}} \stackrel{\text{def}}{=} \\ \quad \left\{ \left(\begin{array}{l} (\boldsymbol{x} \Rrightarrow v * \rho \times a \Mapsto w \cdot u), \\ (\boldsymbol{x} \Rrightarrow u * \rho \times a \Mapsto w \cdot u) \end{array} \right) \;\middle|\; \begin{array}{r} a = \mathcal{E}\,[\![E_1]\!]\,(\boldsymbol{x} \Rrightarrow v * \rho) \text{ and} \\ w = \mathcal{E}\,[\![E_2]\!]\,(\boldsymbol{x} \Rrightarrow v * \rho) \end{array} \right\} \cup \\ \quad \left\{ \left(\begin{array}{l} (\boldsymbol{x} \Rrightarrow v * \rho \times a \Mapsto [l \cdot w]), \\ (\boldsymbol{x} \Rrightarrow \boldsymbol{nil} * \rho \times a \Mapsto [l \cdot w]) \end{array} \right) \;\middle|\; \begin{array}{r} a = \mathcal{E}\,[\![E_1]\!]\,(\boldsymbol{x} \Rrightarrow v * \rho) \text{ and} \\ w = \mathcal{E}\,[\![E_2]\!]\,(\boldsymbol{x} \Rrightarrow v * \rho) \end{array} \right\}. \end{array}$$

Fix arbitrary $a \in \mathsf{Addr}$, $w, u \in \mathsf{Val}$, $l \in \mathsf{Val}^*$ and $c \in \mathsf{C}_{\mathsf{LStore}}$. It is sufficient to establish that the procedure body of `getNext` meets the following specifications:

$$\begin{array}{c} \{[\![(\boldsymbol{i} \Rrightarrow a) * (\boldsymbol{w} \Rrightarrow w) * (\boldsymbol{v} \Rrightarrow -) \times (\{c\} \bullet a \Mapsto w \cdot u)]\!]\} \\ [\![\boldsymbol{v} := \boldsymbol{i}.\texttt{getNext}(\boldsymbol{w})]\!] \\ \{[\![(\boldsymbol{i} \Rrightarrow a) * (\boldsymbol{w} \Rrightarrow w) * (\boldsymbol{v} \Rrightarrow u) \times (\{c\} \bullet a \Mapsto w \cdot u)]\!]\} \end{array} \tag{4}$$

$$\begin{array}{c} \{[\![(\boldsymbol{i} \Rrightarrow a) * (\boldsymbol{w} \Rrightarrow w) * (\boldsymbol{v} \Rrightarrow -) \times (\{c\} \bullet a \Mapsto [l \cdot w])]\!]\} \\ [\![\boldsymbol{v} := \boldsymbol{i}.\texttt{getNext}(\boldsymbol{w})]\!] \\ \{[\![(\boldsymbol{i} \Rrightarrow a) * (\boldsymbol{w} \Rrightarrow w) * (\boldsymbol{v} \Rrightarrow \boldsymbol{nil}) \times (\{c\} \bullet a \Mapsto [l \cdot w])]\!]\} \end{array}. \tag{5}$$

Either specification (4) holds trivially, since the precondition is equivalent to false, or $c = a \Mapsto [l_1 \cdot \_ \cdot l_2] * ls$ for some $l_1, l_2 \in \mathsf{Lst}$ with w and u not in either l_1 or l_2, and some

$E.\texttt{value} \stackrel{\text{def}}{=} E$

$x := \texttt{newNode}() \stackrel{\text{def}}{=} x := \texttt{alloc}(2)$

$\texttt{disposeNode}(x) \stackrel{\text{def}}{=} \texttt{dispose}(x, 2)$

```
v := getHead(i) {
  local x in
    x := [i];
    if x = nil then
      v := x
    else
      v := [x.value]
}

v := getTail(i) {
  local x, y in
    x := [i];
    if x = nil then
      v := x
    else
      y := [x.next];
      while y ≠ nil do
        x := y;
        y := [x.next]   ;
      v := [x.value]
}

v := getNext(i, w) {
  local x in
    x := [i];
    v := [x.value];
    while v ≠ w do
      x := [x.next];
      v := [x.value]   ;
    x := [x.next];
    if x = nil then
      v := x
    else
      v := [x.value]
}
```

$E.\texttt{next} \stackrel{\text{def}}{=} E + 1$

$x := \texttt{newRoot}() \stackrel{\text{def}}{=} x := \texttt{alloc}(1)$

$\texttt{disposeRoot}(x) \stackrel{\text{def}}{=} \texttt{dispose}(x, 1)$

```
v := getPrev(i, w) {
  local x, y in
    x := [i];
    v := [x.value];
    if v = w then
      v := nil
    else
      while v ≠ w do
        y := x;
        x := [y.next];
        v := [x.value]   ;
      v := [y.value]
}

v := pop(i) {
  local x, y in
    x := [i];
    if x = nil then
      v := x
    else
      y := [x.next];
      [i] := y;
      v := [x.value];
      disposeNode(x)
}

push(i, v) {
  local X, y in
    x := newNode();
    [x.value] := v;
    y := [i];
    [x.next] := y;
    [i] := x
}
```

Fig. 10. Linked-list-based list store implementation

```
remove(i, v) {
  local u, x, y, z in
    x := [i];
    u := [x.value];
    y := [x.next];
    if u = v then
      [i] := y;
      disposeNode(x)
    else
      u := [y.value];
      while u ≠ v do
        x := y;
        y := [x.next];
        u := [y.value]   ;
      z := [y.next];
      [x.next] := z;
      disposeNode(y)
}

i := newList() {
  i := newRoot();
  [i] := nil
}
```

```
insert(i, v, w) {
  local u, x, y, z in
    x := [i];
    u := [x.value];
    while u ≠ v do
      x := [x.next];
      u := [x.value]   ;
    y := [x.next];
    z := newNode();
    [z.value] := w;
    [z.next] := y;
    [x.next] := z
}

deleteList(i) {
  local x, y in
    x := [i];
    while x ≠ nil do
      y := x;
      x := [y.next];
      disposeNode(y)   ;
    disposeRoot(i)
}
```

Fig. 11. Linked-list-based list store implementation

$ls \in \mathsf{LStore}$. A proof outline for this case is given in Fig. 13. Similarly, either specification (5) holds trivially or $c = ls$ for some $ls \in \mathsf{LStore}$. A proof outline for this case is given in Fig. 14. In both cases the `getNext` implementation performs the same search for the value w in the list. The proof outline for this common part is given in Fig. 12.

7. Conclusions

We have shown how to refine module specifications, given by abstract local reasoning, into correct implementations. This provides a justification for the soundness of abstract local reasoning with context algebras. We have identified two general approaches for proving the correctness of an implementation with respect to an abstract specification: locality-preserving and locality-breaking translations. Locality-preserving translations relate the abstract locality of a module with the low-level locality of its implementation. This is subtle since disjoint structures at the high-level are not quite disjoint at the low-level, because of the additional crust that is required to handle the pointer surgery. Locality-preserving translations thus establish a "fiction of disjointness" at the abstract level. Meanwhile, locality-breaking translations establish a "fiction of locality", by justifying abstract locality even though this locality is not matched by the implementation.

We now conclude with some discussion of our choices and of the possible future extensions of our work.

$$
\begin{array}{l}
\left\{\exists p.\, \boldsymbol{i} \Rightarrow a * \boldsymbol{w} \Rightarrow w * \boldsymbol{v} \Rightarrow - * \boldsymbol{x} \Rightarrow - \times a \mapsto p * \langle\!\langle l_1 \cdot w \rangle\!\rangle^{(p,y)}\right\} \\
\boldsymbol{x} := [\boldsymbol{i}]\,; \\
\left\{\exists p.\, \boldsymbol{i} \Rightarrow a * \boldsymbol{w} \Rightarrow w * \boldsymbol{v} \Rightarrow - * \boldsymbol{x} \Rightarrow p \times a \mapsto p * \langle\!\langle l_1 \cdot w \rangle\!\rangle^{(p,y)}\right\} \\
\left\{\begin{array}{l} \exists p.\, \boldsymbol{w} \Rightarrow w * \boldsymbol{v} \Rightarrow - * \boldsymbol{x} \Rightarrow p \times \\ \left((l_1 = \varnothing \wedge p \mapsto w, y) \vee (\exists v, l_1', q.\, l_1 = v \cdot l_1'' \wedge p \mapsto w, q * \langle\!\langle l_1'' \cdot w \rangle\!\rangle^{(q,y)})\right) \end{array}\right\} \\
\boldsymbol{v} := [\boldsymbol{x}.\mathtt{value}]\,; \\
\left\{\begin{array}{l} \exists v, x, l_1', l_1'', q.\, \boldsymbol{w} \Rightarrow w * \boldsymbol{v} \Rightarrow v * \boldsymbol{x} \Rightarrow x \times \\ l_1 \cdot w = l_1' \cdot v \cdot l_1'' \wedge \langle\!\langle l_1' \rangle\!\rangle^{(p,x)} * x \mapsto v, q * \langle\!\langle l_1'' \rangle\!\rangle^{(q,y)} \end{array}\right\} \\
\mathtt{while}\ \boldsymbol{v} \neq \boldsymbol{w}\ \mathtt{do} \\
\quad \left\{\begin{array}{l} \exists v, v', x, x', l_1', l_1'', q.\, \boldsymbol{w} \Rightarrow w * \boldsymbol{v} \Rightarrow v * \boldsymbol{x} \Rightarrow x \times \\ l_1 \cdot w = l_1' \cdot v \cdot v' \cdot l_1'' \wedge \langle\!\langle l_1' \rangle\!\rangle^{(p,x)} * x \mapsto v, x' * x' \mapsto v', q * \langle\!\langle l_1'' \rangle\!\rangle^{(q,y)} \end{array}\right\} \\
\quad \boldsymbol{x} := [\boldsymbol{x}.\mathtt{next}]\,; \\
\quad \boldsymbol{v} := [\boldsymbol{x}.\mathtt{value}] \\
\quad \left\{\begin{array}{l} \exists v, v', x, x', l_1', l_1'', q.\, \boldsymbol{w} \Rightarrow w * \boldsymbol{v} \Rightarrow v' * \boldsymbol{x} \Rightarrow x' \times \\ l_1 \cdot w = l_1' \cdot v \cdot v' \cdot l_1'' \wedge \langle\!\langle l_1' \rangle\!\rangle^{(p,x)} * x \mapsto v, x' * x' \mapsto v', q * \langle\!\langle l_1'' \rangle\!\rangle^{(q,y)} \end{array}\right\} \\
\quad \left\{\begin{array}{l} \exists v, x, l_1', l_1'', q.\, \boldsymbol{w} \Rightarrow w * \boldsymbol{v} \Rightarrow v * \boldsymbol{x} \Rightarrow x \times \\ l_1 \cdot w = l_1' \cdot v \cdot l_1'' \wedge \langle\!\langle l_1' \rangle\!\rangle^{(p,x)} * x \mapsto v, q * \langle\!\langle l_1'' \rangle\!\rangle^{(q,y)} \end{array}\right\}\ ; \\
\left\{\exists x.\, \boldsymbol{w} \Rightarrow w * \boldsymbol{v} \Rightarrow w * \boldsymbol{x} \Rightarrow x \times \langle\!\langle l_1 \rangle\!\rangle^{(p,x)} * x \mapsto w, y\right\} \\
\boldsymbol{x} := [\boldsymbol{x}.\mathtt{next}] \\
\left\{\boldsymbol{w} \Rightarrow w * \boldsymbol{v} \Rightarrow w * \boldsymbol{x} \Rightarrow y \times \langle\!\langle l_1 \cdot w \rangle\!\rangle^{(p,y)}\right\} \\
\left\{\exists p.\, \boldsymbol{i} \Rightarrow a * \boldsymbol{w} \Rightarrow w * \boldsymbol{v} \Rightarrow w * \boldsymbol{x} \Rightarrow y \times a \mapsto p * \langle\!\langle l_1 \cdot w \rangle\!\rangle^{(p,y)}\right\}
\end{array}
$$

Fig. 12. Proof outline for search part of `getNext` implementation (common part)

7.1. *On Locality-Preserving versus Locality-Breaking*

Despite the fact that their names imply a black-and-white distinction between locality-preserving and locality-breaking translations, there is no clear distinction between where the two techniques are applicable.

As an example, consider the implementation of the list module from §6.3. We proved that this implementation gave a sound module translation using the locality-breaking technique, since some of the basic operations have a low-level footprint that incorporates an arbitrarily large portion of the linked list. However, it is quite reasonable to identify portions of the low-level state (individual nodes in a linked list) that correspond to portions of the high level state (individual elements of the corresponding list). In fact, the locality-preserving approach can be applied to this implementation, by treating the portion of the linked list that leads up to the nodes of interest as crust.

On the other hand, consider the implementation of the tree module from §5.5. In this case, the crust could be arbitrarily large, accounting for all of the siblings of the top level of the tree. This suggests that the locality-breaking approach is also applicable.

Clearly, there is a significant overlap in the applicability of the two approaches, but it turns out that there are no cases in which one approach is applicable but the other is not.

$$
\begin{array}{l}
v := \texttt{getNext}(i, w)\ \{ \\
\quad \left\{\ [\![i \Rrightarrow a * w \Rrightarrow w * v \Rrightarrow - \times a \Mapsto [l_1 \cdot w \cdot u \cdot l_2] * ls]\!]\ \right\} \\
\quad \left\{\ \exists y, z, p.\ i \Rrightarrow a * w \Rrightarrow w * v \Rrightarrow - \times a \mapsto p * \langle\!\langle l_1 \cdot w \rangle\!\rangle^{(p,y)} * y \mapsto u, z * \langle\!\langle l_2 \rangle\!\rangle^{(z,nil)} * (\!| ls |\!)\ \right\} \\
\qquad \left\{\ \exists p.\ i \Rrightarrow a * w \Rrightarrow w * v \Rrightarrow - \times a \mapsto p * \langle\!\langle l_1 \cdot w \rangle\!\rangle^{(p,y)} * y \mapsto u, z\ \right\} \\
\qquad \texttt{local } x \texttt{ in} \\
\qquad\quad \left\{\ \exists p.\ i \Rrightarrow a * w \Rrightarrow w * v \Rrightarrow - * x \Rrightarrow - \times a \mapsto p * \langle\!\langle l_1 \cdot w \rangle\!\rangle^{(p,y)} * y \mapsto u, z\ \right\} \\
\qquad\qquad \text{(see Fig. 12)} \\
\qquad\quad \left\{\ \exists p.\ i \Rrightarrow a * w \Rrightarrow w * v \Rrightarrow w * x \Rrightarrow y \times a \mapsto p * \langle\!\langle l_1 \cdot w \rangle\!\rangle^{(p,y)} * y \mapsto u, z\ \right\} \\
\qquad\quad \texttt{if } x = nil \texttt{ then} \\
\qquad\qquad v := x \\
\qquad\quad \texttt{else} \\
\qquad\qquad v := [x.\texttt{value}] \\
\qquad\quad \left\{\ \exists p.\ i \Rrightarrow a * w \Rrightarrow w * v \Rrightarrow u * x \Rrightarrow y \times a \mapsto p * \langle\!\langle l_1 \cdot w \rangle\!\rangle^{(p,y)} * y \mapsto u, z\ \right\} \\
\qquad \left\{\ \exists p.\ i \Rrightarrow a * w \Rrightarrow w * v \Rrightarrow u \times a \mapsto p * \langle\!\langle l_1 \cdot w \rangle\!\rangle^{(p,y)} * y \mapsto u, z\ \right\} \\
\quad \left\{\ \exists y, z, p.\ i \Rrightarrow a * w \Rrightarrow w * v \Rrightarrow u \times a \mapsto p * \langle\!\langle l_1 \cdot w \rangle\!\rangle^{(p,y)} * y \mapsto u, z * \langle\!\langle l_2 \rangle\!\rangle^{(z,nil)} * (\!| ls |\!)\ \right\} \\
\quad \left\{\ [\![i \Rrightarrow a * w \Rrightarrow w * v \Rrightarrow u \times a \Mapsto [l_1 \cdot w \cdot u \cdot l_2] * ls]\!]\ \right\} \\
\}
\end{array}
$$

Fig. 13. Proof outline for `getNext` implementation (success case)

A locality-preserving translation can be used to construct a locality-breaking translation and vice versa.

From locality-preserving to locality-breaking is simple. Assume that $\tau : \mathbb{A} \to \mathbb{B}$ is a locality-preserving translation. For all $c \in \mathsf{C}_{\mathbb{A}}$, $\varphi \in \mathsf{Cmd}_{\mathbb{A}}$ and $(P, Q) \in \textsc{Ax}\,[\![\varphi]\!]_{\mathbb{A}}$, there is a derivation of $\vdash_{\mathbb{A}} \{\{\varnothing, c\} \bullet P\}\ \varphi\ \{\{\varnothing, c\} \bullet Q\}$ which simply uses Axiom followed by Frame. By the soundness of locality-preserving translations, there must therefore also be a derivation of $\vdash_{\mathbb{B}} \{[\![\{\varnothing, c\} \bullet P]\!]_\tau\}\ [\![\varphi]\!]_\tau\ \{[\![\{\varnothing, c\} \bullet Q]\!]_\tau\}$. Thus τ defines a locality-breaking translation.

From locality-breaking to locality-preserving is slightly trickier. Assume that $\tau : \mathbb{A} \to \mathbb{B}$ is a locality-breaking translation. For the interface sets, choose $\mathcal{I}_{\text{in}} = \{1\}$ and $\mathcal{I}_{\text{out}} = \mathsf{C}_{\mathbb{A}}$. Define the representation functions for data $\chi_{\mathbb{A}}$ and contexts c as follows:

$$
\langle\!\langle \chi_{\mathbb{A}} \rangle\!\rangle^{c} = \{\chi_{\mathbb{B}} \mid \chi_{\mathbb{B}}\ \alpha_\tau\ (c \bullet \chi_{\mathbb{A}})\}
$$

$$
\langle\!\langle c \rangle\!\rangle^{c_1}_{c_2} = \begin{cases} \mathbf{I}_{\mathbb{B}} & \text{if } c_2 = c_1 \bullet c \\ \emptyset & \text{otherwise.} \end{cases}
$$

Finally, choose $\mathcal{F} = \{1\}$ and define $\Cap_c = \mathbf{I}_{\mathbb{B}}$. Application preservation holds by construction, as does crust inclusion. Axiom correctness simply reduces to the criterion that τ is a locality-breaking translation, *i.e.* that the axioms, with each singleton frame, hold under translation. Thus τ defines a locality-preserving translation.

$$
\begin{array}{l}
\boldsymbol{v} := \texttt{getNext}(\boldsymbol{i}, w)\ \{ \\
\quad \left\{\ [\![\boldsymbol{i} \Rightarrow a * w \Rightarrow w * \boldsymbol{v} \Rightarrow - \times a \Rightarrow [l \cdot w] * ls]\!]\ \right\} \\
\quad \left\{\ \exists p.\, \boldsymbol{i} \Rightarrow a * w \Rightarrow w * \boldsymbol{v} \Rightarrow - \times a \mapsto p * \langle\!\langle l \cdot w \rangle\!\rangle^{(p, \boldsymbol{nil})} * (\!| ls |\!)\ \right\} \\
\qquad \left\{\ \exists p.\, \boldsymbol{i} \Rightarrow a * w \Rightarrow w * \boldsymbol{v} \Rightarrow - \times a \mapsto p * \langle\!\langle l \cdot w \rangle\!\rangle^{(p, \boldsymbol{nil})}\ \right\} \\
\qquad \texttt{local}\ \boldsymbol{x}\ \texttt{in} \\
\qquad\quad \left\{\ \exists p.\, \boldsymbol{i} \Rightarrow a * w \Rightarrow w * \boldsymbol{v} \Rightarrow - * \boldsymbol{x} \Rightarrow - \times a \mapsto p * \langle\!\langle l \cdot w \rangle\!\rangle^{(p, \boldsymbol{nil})}\ \right\} \\
\qquad\qquad \text{(see Fig. 12)} \\
\qquad\quad \left\{\ \exists p.\, \boldsymbol{i} \Rightarrow a * w \Rightarrow w * \boldsymbol{v} \Rightarrow w * \boldsymbol{x} \Rightarrow \boldsymbol{nil} \times a \mapsto p * \langle\!\langle l \cdot w \rangle\!\rangle^{(p, \boldsymbol{nil})}\ \right\} \\
\qquad\quad \texttt{if}\ \boldsymbol{x} = \boldsymbol{nil}\ \texttt{then} \\
\qquad\qquad \boldsymbol{v} := \boldsymbol{x} \\
\qquad\quad \texttt{else} \\
\qquad\qquad \boldsymbol{v} := [\boldsymbol{x}.\texttt{value}] \\
\qquad\quad \left\{\ \exists p.\, \boldsymbol{i} \Rightarrow a * w \Rightarrow w * \boldsymbol{v} \Rightarrow \boldsymbol{nil} * \boldsymbol{x} \Rightarrow \boldsymbol{nil} \times a \mapsto p * \langle\!\langle l \cdot w \rangle\!\rangle^{(p, \boldsymbol{nil})}\ \right\} \\
\qquad \left\{\ \exists p.\, \boldsymbol{i} \Rightarrow a * w \Rightarrow w * \boldsymbol{v} \Rightarrow \boldsymbol{nil} \times a \mapsto p * \langle\!\langle l \cdot w \rangle\!\rangle^{(p, \boldsymbol{nil})}\ \right\} \\
\quad \left\{\ \exists p.\, \boldsymbol{i} \Rightarrow a * w \Rightarrow w * \boldsymbol{v} \Rightarrow \boldsymbol{nil} \times a \mapsto p * \langle\!\langle l \cdot w \rangle\!\rangle^{(p, \boldsymbol{nil})} * (\!| ls |\!)\ \right\} \\
\quad \left\{\ [\![\boldsymbol{i} \Rightarrow a * w \Rightarrow w * \boldsymbol{v} \Rightarrow \boldsymbol{nil} \times a \Rightarrow [l \cdot w] * ls]\!]\ \right\} \\
\}
\end{array}
$$

Fig. 14. Proof outline for `getNext` implementation (failure case)

Note that the transformation from locality-breaking to locality-preserving translations is not a perfect inverse of the transformation in the other direction. This is because locality-breaking transformations do not embody the notion of representing a substructure, and therefore it is necessary to define the representation functions by providing the *entire* enveloping context as the interface.

7.2. *On Abstract Predicates*

One way of viewing the translation functions is as abstract predicates: that is, $[\![P]\!]$ is an abstract predicate parametrised by P. However, viewing this as a completely abstract entity does not confer abstract local reasoning. Exposing such axioms as $[\![P]\!] \vee [\![Q]\!] \leftrightarrow [\![P \vee Q]\!]$ is a start, it allows the low-level disjunction rule to implement its high-level counterpart, and is possible with some formulations of abstract predicates (Dinsdale-Young et al., 2010a). However, abstract predicates do not currently provide a mechanism for exporting meta-theorems such as the soundness of the abstract frame rule. That is to say, there is no way to expose the fact that if $\{[\![P]\!]\}\ \mathbb{C}\ \{[\![Q]\!]\}$ holds then so does $\{[\![K \bullet P]\!]\}\ \mathbb{C}\ \{[\![K \bullet P]\!]\}$. This work suggests that including such a mechanism could be a valuable addition to the abstract predicate methodology.

7.3. *On Concurrency*

Extending the results presented here to a concurrent setting is not entirely trivial. A separation logic-style parallel rule is only appropriate when the model has a commutative $*$-like operator; while the heap and list modules have such an operator, the tree model does not. (The segment logic of Gardner and Wheelhouse (2009) is a promising approach to introducing a commutative $*$ to structured models such as trees.)

Assume that we have a sound module translation from an abstract module to a concrete one, and that both models include a commutative $*$ connective, such that, for every state d there is some context c such that, for all states d', $d * d' = c \bullet d'$. If the implementations of the module operations behave as atomic operations, then the separation logic parallel rule should be sound at the abstract level. That is, if

$$\{[\![P_1]\!]\}\ [\![\mathbb{C}_1]\!]\ \{[\![Q_1]\!]\}$$
$$\{[\![P_2]\!]\}\ [\![\mathbb{C}_2]\!]\ \{[\![Q_2]\!]\}$$

then

$$\{[\![P_1 * P_2]\!]\}\ [\![\mathbb{C}_1 \parallel \mathbb{C}_2]\!]\ \{[\![Q_1 * Q_2]\!]\}\,.$$

Intuitively, this is because the operations of the threads are effectively interleaved, and so the state of the other thread at each point can be viewed as a frame.

An alternative appproach is that of Dinsdale-Young, Dodds, Gardner, Parkinson, and Vafeiadis (2010a) on *concurrent abstract predicates* (*CAP*). CAP presents a fiction of disjointness for concurrent program modules by building abstractions on top of a powerful permission system. This makes it possible to express sophisticated notions of abstract resource, akin to those we consider here. Ultimately, we hope to reach a unified theory of abstraction and refinement that supports fictions of locality and disjointness in both the sequential and concurrent settings.

References

Cristiano Calcagno, Philippa Gardner, and Uri Zarfaty. Context Logic and tree update. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 271–282, New York, NY, USA, 2005. ACM Press. ISBN 1-58113-830-X. .

Cristiano Calcagno, Philippa Gardner, and Uri Zarfaty. Local reasoning about data update. *Electron. Notes Theor. Comput. Sci.*, 172:133–175, 2007a. ISSN 1571-0661. .

Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. Local action and abstract separation logic. In *LICS '07*, pages 366–378, Washington, DC, USA, 2007b. IEEE Computer Society.

Willem-Paul de Roever and Kai Engelhardt. *Data Refinement: Model-Oriented Proof Methods and Their Comparison*. Cambridge University Press, Cambridge, UK, 1999. ISBN 0521641705.

Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. Concurrent abstract predicates. In Theo D'Hondt, editor, *ECOOP*,

volume 6183 of *Lecture Notes in Computer Science*, pages 504–528. Springer, 2010a. ISBN 978-3-642-14106-5.

Thomas Dinsdale-Young, Philippa Gardner, and Mark Wheelhouse. Abstraction and refinement for local reasoning. Technical report, Imperial College London, 2010b. URL `http://www.doc.ic.ac.uk/~td202/papers/alrfull.pdf`.

Philippa Gardner and Mark J. Wheelhouse. Small specifications for tree update. In Cosimo Laneve and Jianwen Su, editors, *WS-FM*, volume 6194 of *Lecture Notes in Computer Science*, pages 178–195. Springer, 2009. ISBN 978-3-642-14457-8.

Philippa A. Gardner, Gareth D. Smith, Mark J. Wheelhouse, and Uri D. Zarfaty. Local hoare reasoning about dom. In *PODS '08*, pages 261–270, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-108-8. .

Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *TOPLAS*, 12(3):463–492, 1990. ISSN 0164-0925. .

C. A. R. Hoare. Proof of correctness of data representations. *Acta Inf.*, 1(4):271–281, 1972.

Samin S. Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In *POPL 2001*, New York, 2001. ACM Press.

Ivana Mijajlović, Noah Torp-Smith, and Peter W. O'Hearn. Refinement and separation contexts. In *FSTTCS '04*, pages 421–433, Berlin/Heidelberg, Germany, 2004. Springer.

Peter W. O'Hearn, John Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *CSL*, Lecture Notes in Computer Science. Springer, 2001.

M. J. Parkinson and G. M. Bierman. Separation logic and abstraction. In *POPL*, pages 247–258, 2005.

Mohammad Raza and Philippa Gardner. Footprints in local reasoning. *Logical Methods in Computer Science*, 5(2), 2009.

John C. Reynolds. Separation Logic: A logic for shared mutable data structures. In *LICS '02: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1483-9.