Abstraction and Refinement for Local Reasoning

Thomas Dinsdale-Young, Philippa Gardner, and Mark Wheelhouse

Imperial College London
{td202, pg, mjw03}@ic.ac.uk

Abstract. 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 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.

1 Introduction

Traditional Hoare logic is an important tool for proving the correctness of programs. However, with heap programs, it is not possible to use this reasoning in a modular way. This is because it is necessary to account for the possibility of multiple references to the same data. For example, a proof that a program reverses a list cannot be used to establish that a second, disjoint list is unchanged; disjointness conditions must be explicitly added at every step in the proof.

Building on Hoare logic, O'Hearn, Reynolds and Yang addressed this problem by introducing separation logic [12] for reasoning *locally* about heap programs. 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. Separation logic achieves local reasoning by treating state as resource. 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.

Abstraction and refinement 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 essential for modularity because it means that a program can be replaced by any other program that meets the same specification. Abstraction

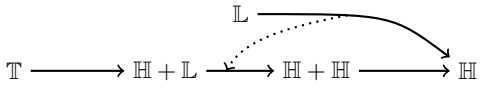

Fig. 1. Module Translations

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 have used abstract predicates to provide abstraction for separation logic [13]. 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 introduced context logic [2], a generalisation of separation logic, to provide such *abstract local reasoning* about structured data. Context logic has been used to reason about programs that manipulate e.g. sequences, multisets and trees [3]. In particular, it has been successfully applied to reason about the W3C DOM tree update library [8]. 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 about structured data with data refinement [10, 5] in order to refine such abstract local specifications into correct implementations.

Mijajlović, Torp-Smith and O'Hearn previously combined data refinement with local operational reasoning [11] to demonstrate that module implementations are equivalent for well-behaved clients, specifically dealing with aliasing issues in the refinement setting. By contrast, we relate axiomatic abstract local reasoning about a module with axiomatic reasoning about its implementations.

The motivating example of this paper is the stepwise refinement of a tree module $\mathbb{T}$. The refinement is illustrated in Fig. 1. We show how the tree module $\mathbb{T}$ may be correctly implemented using the familiar separation-logic heap module $\mathbb{H}$ and an abstract list module $\mathbb{L}$. We then show how this list module $\mathbb{L}$ can be correctly implemented in terms of the heap module $\mathbb{H}$. Our approach is modular, so this refinement can be extended with a second instance of the heap module $\mathbb{H}$ (illustrated with a dotted arrow). Finally, we show that the double-heap module $\mathbb{H} + \mathbb{H}$ can be trivially implemented by the heap module $\mathbb{H}$, completing the refinement from the tree module $\mathbb{T}$ to the heap module $\mathbb{H}$. As a contrast, we also

briefly consider a direct refinement of the tree module $\mathbb{T}$ using the heap module $\mathbb{H}$, although the details of this example are given in the full paper [7].

Our development provides two general techniques for verifying module implementations with respect to their local specifications, using *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.

The full proofs of our results may be found in the full version of this paper [7].

2 Preliminaries

We begin by introducing two key concepts: the definition of a *context algebra* to model program state, and an *axiomatic semantics*, based on context algebras, to describe the behaviour of an imperative programming language.

2.1 State Models

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 [4] to more complex data structures. Whereas separation algebras are based on a commutative combination of resource, context algebras are based on non-commutative resource, which is necessary to handle structured data. We will see that many interesting state models fit the pattern of a context algebra.

Definition 1 (Context Algebra). *A* context algebra $\mathcal{A} = (\mathcal{C}, \mathcal{D}, \bullet, \circ, \mathbf{I}, \mathbf{0})$ *comprises:*

- *a non-empty set of state contexts,* $\mathcal{C}$*;*
- *a non-empty set of abstract states,* $\mathcal{D}$*;*
- *a partially-defined associative context composition function,* $\bullet : \mathcal{C} \times \mathcal{C} \rightharpoonup \mathcal{C}$*;*
- *a partially-defined context application function,* $\circ : \mathcal{C} \times \mathcal{D} \rightharpoonup \mathcal{D}$*,*
  with $c_1 \circ (c_2 \circ d) = (c_1 \bullet c_2) \circ d$ *(undefined terms are considered equal);*
- *a distinguished set of identity contexts,* $\mathbf{I} \subseteq \mathcal{C}$*; and*

– *a distinguished set of empty states,* $\mathbf{0} \subseteq \mathcal{D}$*;*

having the following properties: for all $c \in \mathcal{C}$*,* $d \in \mathcal{D}$*, and* $i' \in \mathbf{I}$

– $i \circ d$ *is defined for some* $i \in \mathbf{I}$*, and whenever* $i' \circ d$ *is defined,* $i' \circ d = d$*;*
– *the relation* $\{(c, d) \mid \exists o \in \mathbf{0}.\, c \circ o = d\}$ *is a total surjective function;*
– $i \bullet c$ *is defined for some* $i \in \mathbf{I}$*, and whenever* $i' \bullet c$ *is defined,* $i' \bullet c = c$*;*
– $c \bullet i$ *is defined for some* $i \in \mathbf{I}$*, and whenever* $c \bullet i'$ *is defined,* $c \bullet i' = c$*.*

Example 1. The following are examples of context algebras:

(a) *Heaps* $h \in \mathrm{H}$ are defined as:

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

where $n \in \mathbb{N}^+$ ranges over unique *heap addresses*, $v \in \mathrm{Val}$ 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*, $\mathcal{H} = (\mathrm{H}, \mathrm{H}, *, *, \{\mathrm{emp}\}, \{\mathrm{emp}\})$. All separation algebras [4] can be viewed as context algebras in this way.

(b) *Variable stores* $\sigma \in \Sigma$ are defined as:

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

where $\boldsymbol{x} \in \mathrm{Var}$ ranges over unique *program variables*, $v \in \mathrm{Val}$ ranges over values, and $*$ is associative and commutative with identity emp. Variable stores form the *variable store context algebra*, $\mathcal{V} = (\Sigma, \Sigma, *, *, \{\mathrm{emp}\}, \{\mathrm{emp}\})$.

(c) *Trees* $t \in \mathrm{T}$ and *tree contexts* $c \in \mathrm{C}$ are defined as:

$$t ::= \varnothing \mid n[t] \mid t \otimes t$$
$$c ::= \_ \mid n[c] \mid t \otimes c \mid c \otimes t$$

where $n \in \mathbb{N}^+$ ranges over unique *node identifiers*, and $\otimes$ is associative with identity $\varnothing$. Context composition and application are standard (substituting a tree or context in the hole), and obviously non-commutative. Trees and tree contexts form the *tree context algebra*, $\mathcal{T} = (\mathrm{C}, \mathrm{T}, \bullet, \circ, \{\_\}, \{\varnothing\})$.

(d) Given context algebras, $\mathcal{A}_1$ and $\mathcal{A}_2$, their product $\mathcal{A}_1 \times \mathcal{A}_2$ (defined as one would expect) is also a context algebra. For example, $\mathcal{H} \times \mathcal{V}$ and $\mathcal{T} \times \mathcal{V}$ combine, respectively, heaps and trees with variable stores.

2.2 Predicates

Predicates are either sets of abstract states (denoted p, q) or sets of state contexts (denoted f, g). We do not fix a particular assertion language, although we do use standard logical notation for conjunction, disjunction, negation and quantification. We lift operations on states and contexts to predicates: for instance, $x \mapsto v$ denotes the predicate $\{x \mapsto v\}$; $\exists v.\, x \mapsto v$ denotes $\{x \mapsto v \mid v \in \mathrm{Val}\}$; $p * q$ denotes $\{d_1 * d_2 \mid d_1 \in p \wedge d_2 \in q\}$; the separating application $f \circ p$ denotes $\{c \circ d \mid c \in f \wedge d \in p\}$; and so on. We also use $\prod^*$ to denote iterated $*$. We use set-theoretic notation for predicate membership ($\in$) and containment ($\subseteq$).

2.3 Language Syntax

Our programming language has a simple imperative core with standard constructs for variables, conditionals, iteration, and procedures. We tailor this language to different domains (heaps, trees, *etc.*) by choosing an appropriate set of basic commands for each domain.

Definition 2 (Programming Language). *Given a set of basic commands Φ, ranged over by φ, the language $\mathcal{L}_\Phi$ is defined by the following grammar:*

$$\begin{aligned}\mathbb{C} ::= &\texttt{skip} \mid \varphi \mid \boldsymbol{x} := \boldsymbol{E} \mid \mathbb{C};\mathbb{C} \mid \texttt{if } \boldsymbol{B} \texttt{ then } \mathbb{C} \texttt{ else } \mathbb{C} \mid \texttt{while } \boldsymbol{B} \texttt{ do } \mathbb{C} \mid \\ &\texttt{procs } \overrightarrow{\boldsymbol{r}}_1 := \texttt{f}_1(\overrightarrow{\boldsymbol{x}}_1)\{\mathbb{C}\}, \cdots, \overrightarrow{\boldsymbol{r}}_k := \texttt{f}_k(\overrightarrow{\boldsymbol{x}}_k)\{\mathbb{C}\} \texttt{ in } \mathbb{C} \mid \\ &\texttt{call } \overrightarrow{\boldsymbol{r}} := \texttt{f}(\overrightarrow{\boldsymbol{E}}) \mid \texttt{local } \boldsymbol{x} \texttt{ in } \mathbb{C}\end{aligned}$$

where $\boldsymbol{x}, \boldsymbol{r}, \ldots \in \mathrm{Var}$ range over program variables, $\boldsymbol{E}, \boldsymbol{E}_1, \ldots \in \mathrm{Exp}_{\mathrm{Val}}$ range over value expressions, $\overrightarrow{\boldsymbol{x}}, \overrightarrow{\boldsymbol{E}}, \ldots$ represent vectors of program variables or expressions, $\boldsymbol{B} \in \mathrm{Exp}_{\mathrm{Bool}}$ ranges over boolean expressions, and $\texttt{f}, \texttt{f}_1, \ldots \in \mathrm{PName}$ range over procedure names.

2.4 Axiomatic Semantics

We give the semantics of the language $\mathcal{L}_\Phi$ as a program logic based on local Hoare reasoning. We model the state with the context algebra, $\mathcal{A} \times \mathcal{V}$, which combines two context algebras: the context algebra, $\mathcal{A}$, manipulated only by the commands of Φ; and the variable store context algebra, $\mathcal{V}$, used to interpret program variables. By treating variables as resource [1], we are able to avoid side-conditions in our proof rules. A set of axioms $\mathrm{Ax} \subseteq \mathcal{P}(\mathcal{D}_\mathcal{A} \times \Sigma) \times \Phi \times \mathcal{P}(\mathcal{D}_\mathcal{A} \times \Sigma)$ provides the semantics for the commands of Φ, where $\mathcal{D}_\mathcal{A}$ is the set of abstract states from $\mathcal{A}$ and Σ is the set of variable stores from $\mathcal{V}$.

The judgments of our proof system have the form $\Gamma \vdash \{p\}\ \mathbb{C}\ \{q\}$, where $p, q \in \mathcal{P}(\mathcal{D}_\mathcal{A} \times \Sigma)$ are predicates, $\mathbb{C} \in \mathcal{L}_\Phi$ 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 judgments 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.

The proof rules of the program logic are given in Fig. 2. The semantics of value expressions $[\![\boldsymbol{E}]\!]_\sigma$ is the value of $\boldsymbol{E}$ in variable store σ. The variable store predicate ρ denotes an arbitrary variable store that evaluates all of the program variables that are read but not written in each command under consideration. We write $\mathrm{vars}(\rho)$ and $\mathrm{vars}(\boldsymbol{E})$ to denote the variables in ρ and $\boldsymbol{E}$ respectively.

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 rules Assgn, Local, PDef and PCall are standard, adapted to our treatment of variables as resource. The

$$\frac{(p,\varphi,q) \in \mathrm{Ax}}{\Gamma \vdash \{p\}\ \varphi\ \{q\}}\ \textsc{Axiom} \qquad \frac{\Gamma \vdash \{p\}\ \mathbb{C}\ \{q\}}{\Gamma \vdash \{f \circ p\}\ \mathbb{C}\ \{f \circ q\}}\ \textsc{Frame}$$

$$\frac{\mathrm{vars}(\rho) = \mathrm{vars}(\boldsymbol{E}) - \{\boldsymbol{x}\}}{\Gamma \vdash \{\mathbf{0}_{\mathcal{A}} \times (\boldsymbol{x} \Rrightarrow v * \rho)\}\ \boldsymbol{x} := \boldsymbol{E}\ \{\mathbf{0}_{\mathcal{A}} \times (\boldsymbol{x} \Rrightarrow [\![\boldsymbol{E}]\!]_{(\boldsymbol{x} \Rightarrow v * \rho)} * \rho)\}}\ \textsc{Assgn}$$

$$\frac{\Gamma \vdash \{(\mathbf{I}_{\mathcal{A}} \times \boldsymbol{x} \Rrightarrow -) \circ p\}\ \mathbb{C}\ \{(\mathbf{I}_{\mathcal{A}} \times \boldsymbol{x} \Rrightarrow -) \circ q\} \quad (\mathbf{I}_{\mathcal{A}} \times \boldsymbol{x} \Rrightarrow -) \circ p \neq \emptyset}{\Gamma \vdash \{p\}\ \mathtt{local}\ \boldsymbol{x}\ \mathtt{in}\ \mathbb{C}\ \{q\}}\ \textsc{Local}$$

$$\frac{\forall (\mathtt{f}_i : P \to Q) \in \Gamma.\ \Gamma', \Gamma \vdash \begin{array}{c}\{\exists \overrightarrow{v}.\, P(\overrightarrow{v}) \times (\overrightarrow{\boldsymbol{x}_i} \Rrightarrow \overrightarrow{v} * \overrightarrow{\boldsymbol{r}_i} \Rrightarrow -)\} \\ \mathbb{C}_i \\ \{\exists \overrightarrow{w}.\, Q(\overrightarrow{w}) \times (\overrightarrow{\boldsymbol{x}_i} \Rrightarrow - * \overrightarrow{\boldsymbol{r}_i} \Rrightarrow \overrightarrow{w})\}\end{array} \qquad \forall (\mathtt{f} : P \to Q) \in \Gamma.\, \exists i.\, \mathtt{f} = \mathtt{f}_i \qquad \Gamma', \Gamma \vdash \{p\}\ \mathbb{C}\ \{q\}}{\Gamma' \vdash \{p\}\ \mathtt{procs}\ \overrightarrow{\boldsymbol{r}_1} := \mathtt{f}_1(\overrightarrow{\boldsymbol{x}_1})\{\mathbb{C}_1\}, \ldots, \overrightarrow{\boldsymbol{r}_k} := \mathtt{f}_k(\overrightarrow{\boldsymbol{x}_k})\{\mathbb{C}_k\}\ \mathtt{in}\ \mathbb{C}\ \{q\}}\ \textsc{PDef}$$

$$\frac{\mathrm{vars}(\rho) = \mathrm{vars}(\boldsymbol{E}) - \{\overrightarrow{\boldsymbol{r}}\}}{\Gamma, (\mathtt{f} : P \to Q) \vdash \begin{array}{c}\left\{P([\![\overrightarrow{\boldsymbol{E}}]\!]_{(\overrightarrow{\boldsymbol{r}} \Rightarrow \overrightarrow{v} * \rho)}) \times (\overrightarrow{\boldsymbol{r}} \Rrightarrow \overrightarrow{v} * \rho)\right\} \\ \mathtt{call}\ \overrightarrow{\boldsymbol{r}} := \mathtt{f}(\overrightarrow{\boldsymbol{E}}) \\ \{\exists \overrightarrow{w}.\, Q(\overrightarrow{w}) \times (\overrightarrow{\boldsymbol{r}} \Rrightarrow \overrightarrow{w} * \rho)\}\end{array}}\ \textsc{PCall}$$

Fig. 2. Selected local Hoare logic rules for $\mathcal{L}_{\Phi}$.

Assgn rule not only requires the resource $\boldsymbol{x} \Rrightarrow v$, but also the resource ρ containing the other variables used in $\boldsymbol{E}$. For the Local rule, recall that the predicate p specifies a set of pairs consisting of resource from $\mathcal{D}_{\mathcal{A}}$ and variable resource. The predicate $(\mathbf{I}_{\mathcal{A}} \times \boldsymbol{x} \Rrightarrow -) \circ p$ therefore extends the variable component with variable x of indeterminate initial value. If a local variable block is used to redeclare 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 $\mathtt{f}$ has parametrised predicates $P = \lambda \overrightarrow{x}.p$ and $Q = \lambda \overrightarrow{r}.q$ as its pre- and postcondition, with $P(\overrightarrow{v}) = p[\overrightarrow{v}/\overrightarrow{x}]$ and $Q(\overrightarrow{w}) = q[\overrightarrow{w}/\overrightarrow{r}]$; the parameters carry the call and return values of the procedure. We omit the Cons, Disj, Skip, Seq, If and While rules, which are standard. For all of our examples, the conjunction rule is admissible; in general, this is not the case.

3 Abstract Modules

The language given in §2 and its semantics are parameterised by a context algebra, a set of commands and a set of axioms. These parameters constitute an abstract description of a module. We shall use this notion of an *abstract module* to show how to correctly implement one module in terms of another.

Definition 3 (Abstract Module). *An* abstract module $\mathbb{A} = (\mathcal{A}_{\mathbb{A}}, \Phi_{\mathbb{A}}, \mathrm{Ax}_{\mathbb{A}})$ *consists of a context algebra* $\mathcal{A}_{\mathbb{A}}$ *with abstract state set* $\mathcal{D}_{\mathbb{A}}$*, a set of commands* $\Phi_{\mathbb{A}}$ *and a set of axioms* $\mathrm{Ax}_{\mathbb{A}} \subseteq \mathcal{P}(\mathcal{D}_{\mathbb{A}} \times \Sigma) \times \Phi_{\mathbb{A}} \times \mathcal{P}(\mathcal{D}_{\mathbb{A}} \times \Sigma)$.

Notation. We write $\mathcal{L}_{\mathbb{A}}$ for the language $\mathcal{L}_{\Phi_{\mathbb{A}}}$. We write $\vdash_{\mathbb{A}}$ for the proof judgment determined by the abstract module. When $\mathbb{A}$ can be inferred from context, we may simply write $\vdash$ instead of $\vdash_{\mathbb{A}}$.

3.1 Heap Module

The first and most familiar abstract module we consider is the abstract heap module, $\mathbb{H} = (\mathcal{H}, \Phi_{\mathbb{H}}, \mathrm{Ax}_{\mathbb{H}})$, which extends the core language with standard heap-update commands. The context algebra $\mathcal{H}$ was defined in Example 1. We give the heap update commands in Definition 4, and the axioms for describing the behaviour of these commands in Definition 5.

Definition 4 (Heap Update Commands). *The set of heap update commands $\Phi_{\mathbb{H}}$ comprises: allocation, $\boldsymbol{n} := \mathtt{alloc}(E)$; disposal, $\mathtt{dispose}(E, E')$; mutation, $[E] := E'$; and lookup $\boldsymbol{n} := [E]$.*

Definition 5 (Heap Axioms). *The set of heap axioms $\mathrm{Ax}_{\mathbb{H}}$ comprises:*

$$\left\{\begin{array}{l}\mathrm{emp} \times \boldsymbol{n} \Rrightarrow v * \rho \\ \wedge\, [\![E]\!]_{\rho * \boldsymbol{n} \Rightarrow v} \geq 1\end{array}\right\} \quad \boldsymbol{n} := \mathtt{alloc}(E) \quad \left\{\begin{array}{l}\exists x.\, x \mapsto - * \ldots \\ * \, x + [\![E]\!]_{\rho * \boldsymbol{n} \Rightarrow v} \mapsto - \\ \times \boldsymbol{n} \Rrightarrow x * \rho\end{array}\right\}$$

$$\left\{\begin{array}{l}[\![E]\!]_\rho \mapsto - * \ldots \\ * \, [\![E]\!]_\rho + [\![E']\!]_\rho \mapsto - \times \rho\end{array}\right\} \quad \mathtt{dispose}(E, E') \quad \{\mathrm{emp} \times \rho\}$$

$$\{[\![E]\!]_\rho \mapsto - \times \rho\} \quad [E] := E' \quad \{[\![E]\!]_\rho \mapsto [\![E']\!]_\rho \times \rho\}$$

$$\{[\![E]\!]_{\rho * \boldsymbol{n} \Rightarrow v} \mapsto x \times \boldsymbol{n} \Rrightarrow v * \rho\} \quad \boldsymbol{n} := [E] \quad \{[\![E]\!]_{\rho * \boldsymbol{n} \Rightarrow v} \mapsto x \times \boldsymbol{n} \Rrightarrow x * \rho\}$$

3.2 Tree Module

Another familiar abstract module that we consider is the abstract tree module, $\mathbb{T} = (\mathcal{T}, \Phi_{\mathbb{T}}, \mathrm{Ax}_{\mathbb{T}})$, which extends the core language with tree update commands acting on a single tree, similar to a document in DOM. The tree context algebra $\mathcal{T}$ was defined in Example 1. We give the tree update commands in Definition 6 and their corresponding axioms in Definition 7.

Definition 6 (Tree Update Commands). *The set of tree update commands $\Phi_{\mathbb{T}}$ comprises: relative traversal,* `getUp`, `getLeft`, `getRight`, `getFirst`, `getLast`*; node creation,* `newNodeAfter`*; and subtree deletion* `deleteTree`*.*

Definition 7 (Tree Axioms). *The set of tree update axioms $\mathrm{Ax}_{\mathbb{T}}$ includes:*

$$\left\{\begin{array}{l}[\![E]\!]_{\rho * \boldsymbol{n} \Rightarrow n}[t] \otimes m[t'] \\ \times \boldsymbol{n} \Rrightarrow n * \rho\end{array}\right\} \boldsymbol{n} := \mathtt{getRight}(E) \left\{\begin{array}{l}[\![E]\!]_{\rho * \boldsymbol{n} \Rightarrow n}[t] \otimes m[t'] \\ \times \boldsymbol{n} \Rrightarrow m * \rho\end{array}\right\}$$

$$\left\{\begin{array}{l}m[t' \otimes [\![E]\!]_{\rho * \boldsymbol{n} \Rightarrow n}[t]] \\ \times \boldsymbol{n} \Rrightarrow n * \rho\end{array}\right\} \boldsymbol{n} := \mathtt{getRight}(E) \left\{\begin{array}{l}m[t' \otimes [\![E]\!]_{\rho * \boldsymbol{n} \Rightarrow n}[t]] \\ \times \boldsymbol{n} \Rrightarrow \boldsymbol{null} * \rho\end{array}\right\}$$

$$\left\{\begin{array}{l}[\![E]\!]_{\rho * \boldsymbol{n} \Rightarrow n}[t' \otimes m[t]] \\ \times \boldsymbol{n} \Rrightarrow n * \rho\end{array}\right\} \boldsymbol{n} := \mathtt{getLast}(E) \left\{\begin{array}{l}[\![E]\!]_{\rho * \boldsymbol{n} \Rightarrow n}[t' \otimes m[t]] \\ \times \boldsymbol{n} \Rrightarrow m * \rho\end{array}\right\}$$

$$\left\{\begin{array}{l}[\![E]\!]_{\rho * \boldsymbol{n} \Rightarrow n}[\varnothing] \\ \times \boldsymbol{n} \Rrightarrow n * \rho\end{array}\right\} \boldsymbol{n} := \mathtt{getLast}(E) \left\{\begin{array}{l}[\![E]\!]_{\rho * \boldsymbol{n} \Rightarrow n}[\varnothing] \\ \times \boldsymbol{n} \Rrightarrow \boldsymbol{null} * \rho\end{array}\right\}$$

$$\{[\![E]\!]_\rho[t] \times \rho\}\ \texttt{newNodeAfter}(E)\ \{\exists m.\, [\![E]\!]_\rho[t] \otimes m[\varnothing] \times \rho\}$$
$$\{[\![E]\!]_\rho[t] \times \rho\}\ \ \texttt{deleteTree}(E)\ \ \{\varnothing \times \rho\}$$

The omitted axioms are analogous to those given above.

3.3 List Module

We will study an implementation of the tree module using lists of unique addresses. We therefore define an abstract module for manipulating lists whose elements are unique, $\mathbb{L} = (\mathcal{L}, \Phi_\mathbb{L}, \mathrm{Ax}_\mathbb{L})$. The list context algebra $\mathcal{L}$ is given in Definition 10. The list update commands are given in Definition 11 and their corresponding axioms are given in Definition 12.

Superficially, our abstract list stores resemble heaps, in the sense that we have multiple lists each with a unique address. We write $(i \Mapsto v_1{+}v_2{+}v_3){*}(j \Mapsto w_1{+}v_1)$ to denote a list store consisting of two separate lists $v_1 + v_2 + v_3$ and $w_1 + v_1$, at different addresses i and j. However, unlike heaps, our list store contexts also allow us to consider separation within lists. For example, the same list store can be written as $(i \Mapsto v_1 + - + v_3) \circ (i \Mapsto v_2 * j \Mapsto w_1 + v_1)$, describing a list context $v_1 + - + v_3$ at address i applied to the list store $i \Mapsto v_2 * j \Mapsto w_1 + v_1$: the application puts list v_2 at i into the context hole.

Furthermore, we make a distinction between lists $j \Mapsto w_1 + v_1$ which can be extended by list contexts, and *completed* lists $j \Mapsto [w_1 + v_1]$ which cannot be extended. The reason to work with completed lists is that sometimes we need to know which elements are the first or last elements in a list. For example, the command `getHead` will return the first element of a list, so this element must be fully determined and not subject to change if a frame is applied. Completed lists may be separated into contexts and sublists, for example $j \Mapsto [w_1 + -] \circ j \Mapsto v_1$ is defined, but may not be extended, for example $j \Mapsto w_1 + - \circ j \Mapsto [v_1]$ is undefined.

Definition 8 (List Stores and Contexts). Lists $l \in \mathrm{L}$, list contexts $lc \in \mathrm{Lc}$, list stores $ls \in \mathrm{Ls}$, *and* list store contexts $lsc \in \mathrm{Lsc}$ *are defined by:*

$$l ::= \varepsilon \mid v \mid l + l \qquad ls ::= \mathrm{emp} \mid i \Mapsto l \mid i \Mapsto [l] \mid ls * ls$$
$$lc ::= - \mid lc + l \mid l + lc \qquad lsc ::= ls \mid i \Mapsto lc \mid i \Mapsto [lc] \mid lsc * lsc$$

where $v \in \mathrm{Val}$ *ranges over values, which are taken to occur uniquely in each list or list context,* $i \in \mathrm{Laddr}$ *ranges over list addresses, which are taken to occur uniquely in each list store or list store context,* $+$ *is taken to be associative with identity* ε*, and* $*$ *is taken to be associative and commutative with identity* emp.

Our context application $\circ$ actually subsumes our separating operator $*$, in that as well as extending existing lists, we can also add new lists to the store, for example $(j \Mapsto w_1 + v_1) \circ (i \Mapsto v_1 + v_2 + v_3) = (j \Mapsto w_1 + v_1) * (i \Mapsto v_1 + v_2 + v_3)$.

Definition 9 (Application and Composition). *The application of list store contexts to list stores* $\circ : \mathrm{Lsc} \times \mathrm{Ls} \rightharpoonup \mathrm{Ls}$ *is defined inductively by:*

$$\mathrm{emp} \circ ls = ls$$
$$(lsc * i \Mapsto l) \circ ls = (lsc \circ ls) * i \Mapsto l$$
$$(lsc * i \Mapsto [l]) \circ ls = (lsc \circ ls) * i \Mapsto [l]$$

$$(lsc * i \Mapsto lc) \circ (ls * i \Mapsto l) = (lsc \circ ls) * i \Mapsto lc_{[l/\_]}$$
$$(lsc * i \Mapsto [lc]) \circ (ls * i \Mapsto l) = (lsc \circ ls) * i \Mapsto [lc_{[l/\_]}]$$

where $lc_{[l/\_]}$ denotes the standard replacement of the hole in lc by l. The result of the application is undefined when either the right-hand side is badly formed or no case applies. The composition $\bullet : \textsc{Lsc} \times \textsc{Lsc} \rightharpoonup \textsc{Lsc}$ is defined similarly.

Definition 10 (List-Store Context Algebra). *The* list-store context algebra, $\mathcal{L} = (\textsc{Lsc}, \textsc{Ls}, \bullet, \circ, \{\text{emp}\}, \{\text{emp}\})$ *is given by the above definitions.*

Definition 11 (List Update Commands). *The set of list commands $\Phi_{\mathbb{L}}$ comprises: lookup,* `getHead`, `getTail`, `getNext`, `getPrev`*; stack-style access,* `pop`, `push`*; value removal and insertion,* `remove`, `insert`*; and construction and destruction,* `newList`, `deleteList`*.*

Definition 12 (List Axioms). *The set of list axioms $\textsc{Ax}_{\mathbb{L}}$ includes the following axioms: (the omitted axioms are analogous)*

$$\left\{ \begin{array}{l} [\![E]\!]_{\rho * v \Rightarrow v} \Mapsto [\, v' + l \,] \\ \times\ \boldsymbol{v} \Rrightarrow v * \rho \end{array} \right\}\ \boldsymbol{v} := E.\texttt{getHead}()\ \left\{ \begin{array}{l} [\![E]\!]_{\rho * v \Rightarrow v} \Mapsto [\, v' + l \,] \\ \times\ \boldsymbol{v} \Rrightarrow v' * \rho \end{array} \right\}$$

$$\left\{ \begin{array}{l} [\![E]\!]_{\rho * v \Rightarrow v} \Mapsto [\, \varepsilon \,] \\ \times\ \boldsymbol{v} \Rrightarrow v * \rho \end{array} \right\}\ \boldsymbol{v} := E.\texttt{getHead}()\ \left\{ \begin{array}{l} [\![E]\!]_{\rho * v \Rightarrow v} \Mapsto [\, \varepsilon \,] \\ \times\ \boldsymbol{v} \Rrightarrow \textbf{\textit{null}} * \rho \end{array} \right\}$$

$$\left\{ \begin{array}{l} [\![E]\!]_{\rho * v \Rightarrow v} \Mapsto [\![E']\!]_{\rho * v \Rightarrow v} + u \\ \times\ \boldsymbol{v} \Rrightarrow v * \rho \end{array} \right\}\ \boldsymbol{v} := E.\texttt{getNext}(E')\ \left\{ \begin{array}{l} [\![E]\!]_{\rho * v \Rightarrow v} \Mapsto [\![E']\!]_{\rho * v \Rightarrow v} + u \\ \times\ \boldsymbol{v} \Rrightarrow u * \rho \end{array} \right\}$$

$$\left\{ \begin{array}{l} [\![E]\!]_{\rho * v \Rightarrow v} \Mapsto [\, l + [\![E']\!]_{\rho * v \Rightarrow v} \,] \\ \times\ \boldsymbol{v} \Rrightarrow v * \rho \end{array} \right\}\ \boldsymbol{v} := E.\texttt{getNext}(E')\ \left\{ \begin{array}{l} [\![E]\!]_{\rho * v \Rightarrow v} \Mapsto [\, l + [\![E']\!]_{\rho * v \Rightarrow v} \,] \\ \times\ \boldsymbol{v} \Rrightarrow \textbf{\textit{null}} * \rho \end{array} \right\}$$

$$\{ [\![E]\!]_\rho \Mapsto [\, l \,] \times \rho \wedge ([\![E']\!]_\rho \notin l) \}\quad E.\texttt{push}(E')\quad \{ [\![E]\!]_\rho \Mapsto [\, [\![E']\!]_\rho + l \,] \times \rho \}$$

$$\{ [\![E]\!]_\rho \Mapsto [\![E']\!]_\rho \times \rho \}\quad E.\texttt{remove}(E')\quad \{ [\![E]\!]_\rho \Mapsto \varepsilon \times \rho \}$$

$$\left\{ \begin{array}{l} [\![E]\!]_\rho \Mapsto [\, l + [\![E']\!]_\rho + l' \,] \times \rho \\ \wedge\ ([\![E'']\!]_\rho \notin l + [\![E']\!]_\rho + l') \end{array} \right\}\ E.\texttt{insert}(E', E'')\ \left\{ \begin{array}{l} [\![E]\!]_\rho \Mapsto [\, l + [\![E']\!]_\rho + \\ \qquad [\![E'']\!]_\rho + l' \,] \times \rho \end{array} \right\}$$

$$\{ \emptyset \times \boldsymbol{i} \Rrightarrow i \}\quad \boldsymbol{i} := \texttt{newList}()\quad \{ \exists j.\, j \Mapsto [\, \varepsilon \,] \times \boldsymbol{i} \Rrightarrow j \}$$

$$\{ [\![E]\!]_\rho \Mapsto [\, l \,] \times \rho \}\quad E.\texttt{deleteList}()\quad \{ \emptyset \times \rho \}$$

3.4 Combining Abstract Modules

We provide a natural way of combining abstract modules that enables programs to be written that intermix commands from different modules. For example, we will use the heap and list module combination $\mathbb{H} + \mathbb{L}$ in §5.1 as the basis for implementing $\mathbb{T}$. The combination comprises both commands for manipulating lists and commands for manipulating heaps, defined so that they do not interfere with each other.

Definition 13 (Abstract Module Combination). *Given abstract modules $\mathbb{A}_1 = (\mathcal{A}_{\mathbb{A}_1}, \Phi_{\mathbb{A}_1}, \textsc{Ax}_{\mathbb{A}_1})$ and $\mathbb{A}_2 = (\mathcal{A}_{\mathbb{A}_2}, \Phi_{\mathbb{A}_2}, \textsc{Ax}_{\mathbb{A}_2})$, their combination $\mathbb{A}_1 + \mathbb{A}_2 = (\mathcal{A}_{\mathbb{A}_1} \times \mathcal{A}_{\mathbb{A}_2}, \Phi_{\mathbb{A}_1} \oplus \Phi_{\mathbb{A}_2}, \textsc{Ax}_{\mathbb{A}_1} + \textsc{Ax}_{\mathbb{A}_2})$ is defined by:*

- $\mathcal{A}_{\mathbb{A}_1} \times \mathcal{A}_{\mathbb{A}_2}$ *is the product of context algebras;*
- $\Phi_{\mathbb{A}_1} \oplus \Phi_{\mathbb{A}_2} = (\Phi_{\mathbb{A}_1} \times \{1\}) \cup (\Phi_{\mathbb{A}_2} \times \{2\})$ *is the disjoint union of command sets;*
- $\mathrm{Ax}_{\mathbb{A}_1} + \mathrm{Ax}_{\mathbb{A}_2}$ *is the lifting of the axiom sets* $\mathrm{Ax}_{\mathbb{A}_1}$ *and* $\mathrm{Ax}_{\mathbb{A}_2}$ *using the empty states from* $\mathrm{Ax}_{\mathbb{A}_2}$ *and* $\mathrm{Ax}_{\mathbb{A}_1}$*: formally,* $\mathrm{Ax}_{\mathbb{A}_1} + \mathrm{Ax}_{\mathbb{A}_2} = \{(\pi_1 p, (\varphi, 1), \pi_1 q) \mid (p, \varphi, q) \in \mathrm{Ax}_{\mathbb{A}_1}\} \cup \{(\pi_2 p, (\varphi, 2), \pi_2 q) \mid (p, \varphi, q) \in \mathrm{Ax}_{\mathbb{A}_1}\}$, *where* $\pi_1 p = \{(d,o,\sigma) \mid (d,\sigma) \in p,\ o \in \mathbf{0}_2\}$, $\pi_2 p = \{(o,d,\sigma) \mid (d,\sigma) \in p,\ o \in \mathbf{0}_1\}$.

When the command sets $\Phi_{\mathbb{A}_1}$ and $\Phi_{\mathbb{A}_2}$ are disjoint, we may drop the tags when referring to the commands in the combined abstract module. When we do use the tags, we indicate them with an appropriately placed subscript.

4 Module Translations

We define what it means to correctly implement one module in terms of another, using translations which are reminiscent of downward simulations in [9].

Definition 14 (Sound Module Translation). *A* module translation $\mathbb{A} \to \mathbb{B}$ *from abstract module* $\mathbb{A}$ *to abstract module* $\mathbb{B}$ *consists of*

- *a* state translation *function* $[\![-]\!] : \mathcal{D}_{\mathbb{A}} \to \mathcal{P}(\mathcal{D}_{\mathbb{B}})$*, and*
- *a* substitutive implementation *function* $[\![-]\!] : \mathcal{L}_{\mathbb{A}} \to \mathcal{L}_{\mathbb{B}}$ *obtained by substituting each basic command of* $\Phi_{\mathbb{A}}$ *with a call to a procedure written in* $\mathcal{L}_{\mathbb{B}}$*.*

A module translation is sound *if, for all* $p, q \in \mathcal{P}(\mathcal{D}_{\mathbb{A}} \times \Sigma)$ *and* $\mathbb{C} \in \mathcal{L}_{\mathbb{A}}$*,*

$$\vdash_{\mathbb{A}} \{p\}\ \mathbb{C}\ \{q\} \quad \Longrightarrow \quad \vdash_{\mathbb{B}} \{[\![p]\!]\}\ [\![\mathbb{C}]\!]\ \{[\![q]\!]\}\,.$$

where the predicate translation $[\![-]\!] : \mathcal{P}(\mathcal{D}_{\mathbb{A}} \times \Sigma) \to \mathcal{P}(\mathcal{D}_{\mathbb{B}} \times \Sigma)$ *is the natural lifting of the state translation given by* $[\![p]\!] = \bigvee_{(d,\sigma) \in p} [\![d]\!] \times \sigma$*.*

We will see that sometimes the module structure is preserved by the translations and sometimes it is not; also, sometimes the proof structure is preserved, sometimes not. Notice that, since we are only considering partial correctness, it is always acceptable for the implementation to diverge. In order to make termination guarantees, we could work with total correctness; our decision not to is for simplicity and based on prevailing trends in separation logic and context logic literature [12, 4, 2]. It is possible for our predicate translation to lose information. For instance, if all predicates were unsatisfiable under translation, it would be possible to implement every abstract command with `skip`; such an implementation is useless. It may be desirable to consider some injectivity condition which distinguishes states and predicates of interest. Our results do not rely on this.

Modularity. A translation $\mathbb{A}_1 \to \mathbb{A}_2$ can be naturally lifted to a translation $\mathbb{A}_1 + \mathbb{B} \to \mathbb{A}_2 + \mathbb{B}$, for any module $\mathbb{B}$. We would hope that the resulting translations would be sound, but it is not clear that this holds for all sound translations $\mathbb{A}_1 \to \mathbb{A}_2$. When it is does hold, we say that the translation $\mathbb{A}_1 \to \mathbb{A}_2$ is *modular*. The techniques we consider in this paper provide modular translations, because they inductively transform proofs from module $\mathbb{A}_1$ to proofs in module $\mathbb{A}_2$.

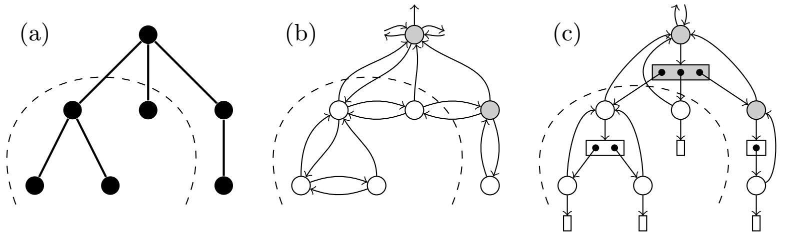

Fig. 3. An abstract tree from T (a), and its representations in H (b) and H × Ls (c).

5 Locality-preserving Translations

Sometimes there is a close correspondence between locality in an abstract module and locality in its implementation. We introduce locality-preserving module translations, and provide a general result that such translations are sound. Recall Fig. 1 of the introduction. We show that module translations $\mathbb{T} \to \mathbb{H} + \mathbb{L}$ and $\mathbb{H} + \mathbb{H} \to \mathbb{H}$ are locality preserving.

Consider Fig. 3 which depicts a simple tree (a), and representations of it in the heap module $\mathbb{H}$ (b), and in the combined heap and list module $\mathbb{H} + \mathbb{L}$ (c). In (b), a node is represented by a memory block of four fields, recording the addresses of the left sibling, parent, right sibling and first child. In (c), a node is represented by a list of the child nodes (shown as boxes) and a block of two fields, recording the address of the parent and the child list. Just as the tree in (a) can be decomposed into a context and a disjoint subtree (as shown by the dashed lined), its representations can also be decomposed: the representations preserve context application. However, we must account for the pointers in the representations which cross the boundary between context and subtree. This means that the representation of a tree must be parameterised by an *interface* to the surrounding context. Similarly, contexts are parameterised by interfaces both to the inner subtree and outer context. We split the interface I into two components: the reference the surrounding context makes *in* to the subtree (the *in* part), and the reference the subtree makes *out* to the surrounding context (the *out* part).

Consider deleting the subtree indicated by the dashed lines in the figure. In the abstract tree, this deletion only operates on the subtree: the axiom for deletion has just the subtree as its precondition. In the implementations however, the deletion also operates on the representation of some of the surrounding context: in (b), this is the parent node and right sibling; in (c), the parent node and child list. We therefore introduce the idea of a *crust* predicate, $\Cap_I^F$, that comprises the minimal additional state required by an implementation. The crust is parameterised by interface I and an additional crust parameter F that fully determine it. In the figure, the crusts for the subtree in (b) and (c) are shown shaded. (In the list-based representation, all the sibling nodes form part of the crust because they are required for node insertion.)

We define locality-preserving translations which incorporate three key properties: *application preservation*, *crust inclusion*, and *axiom correctness*. Application preservation, we have seen, requires that the low-level representations of abstract states can be decomposed in the same manner as the abstract states themselves. Crust inclusion requires that an abstract state's crust is subsumed by any context that is applied (together with the context's own crust). This allows us to frame on arbitrary contexts despite the crust already being present – we simply remove the state's crust from the context before applying it. (Since the crust represents an effective overlap between what represents a state and what represents its context, the abstract view that the two are disjoint is really a fiction of disjointness.) Finally, axiom correctness requires that the implementations of the basic commands meet the specifications given by the abstract module's axioms.

Definition 15 (Locality-Preserving Translation). *For interface set* $\mathcal{I} = \mathcal{I}_{\text{in}} \times \mathcal{I}_{\text{out}}$ *and crust parameter set* $\mathcal{F}$*, a* locality-preserving translation $\mathbb{A} \to \mathbb{B}$ *comprises:*

- *representation functions* $\langle\!\langle - \rangle\!\rangle^{-} : \mathcal{D}_{\mathbb{A}} \times \mathcal{I} \to \mathcal{P}(\mathcal{D}_{\mathbb{B}})$ *and* $\langle\!\langle - \rangle\!\rangle^{-}_{-} : \mathcal{C}_{\mathbb{A}} \times \mathcal{I} \times \mathcal{I} \to \mathcal{P}(\mathcal{C}_{\mathbb{B}})$*;*
- *a crust predicate* $\Subset^F_I$*, parameterised by* $I \in \mathcal{I}$ *and* $F \in \mathcal{F}$*; and*
- *a substitutive implementation function* $[\![-]\!] : \mathcal{L}_{\mathbb{A}} \to \mathcal{L}_{\mathbb{B}}$*,*

for which the following properties hold:

1. **application preservation:** *for all* $f \in \mathcal{P}(\mathcal{C}_{\mathbb{A}})$*,* $p \in \mathcal{P}(\mathcal{D}_{\mathbb{A}})$ *and* $I \in \mathcal{I}$*,*

$$\langle\!\langle f \circ_{\mathbb{A}} p \rangle\!\rangle^{I} \quad = \quad \exists I'.\, \langle\!\langle f \rangle\!\rangle^{I}_{I'} \circ_{\mathbb{B}} \langle\!\langle p \rangle\!\rangle^{I'};$$

2. **crust inclusion:** *for all* $\overrightarrow{out}', \overrightarrow{out} \in \mathcal{I}_{\text{out}}$*,* $F \in \mathcal{F}$*,* $c \in \mathcal{C}_{\mathbb{A}}$*, there exist* $f \in \mathcal{P}(\mathcal{C}_{\mathbb{B}})$*,* $F' \in \mathcal{F}$ *such that, for all* $\overrightarrow{in} \in \mathcal{I}_{\text{in}}$*,*

$$\left(\exists \overrightarrow{in}'.\, \Subset^F_{\overrightarrow{in}', \overrightarrow{out}'} \bullet \langle\!\langle c \rangle\!\rangle^{\overrightarrow{in}', \overrightarrow{out}'}_{\overrightarrow{in}, \overrightarrow{out}}\right) = f \bullet \Subset^{F'}_{\overrightarrow{in}, \overrightarrow{out}}; \text{ and}$$

3. **axiom correctness:** *for all* $(p, \varphi, q) \in \text{Ax}_{\mathbb{A}}$*,* $\overrightarrow{out} \in \mathcal{I}_{\text{out}}$ *and* $F \in \mathcal{F}$*,*

$$\vdash_{\mathbb{B}} \left\{ (\!| p |\!)^{\overrightarrow{out}, F} \right\} \; [\![\varphi]\!] \; \left\{ (\!| q |\!)^{\overrightarrow{out}, F} \right\},$$

where $(\!| p |\!)^{\overrightarrow{out}, F} = \bigvee_{(d,\sigma) \in p} (\exists \overrightarrow{in}.\, \Subset^F_{\overrightarrow{in}, \overrightarrow{out}} \circ \langle\!\langle d \rangle\!\rangle^{\overrightarrow{in}, \overrightarrow{out}}) \times \sigma.$

Notice that this locality-preserving translation is a module translation, with the state translation function $[\![-]\!] : \mathcal{D}_{\mathbb{A}} \to \mathcal{P}(\mathcal{D}_{\mathbb{B}})$ defined by $[\![d]\!] = \exists \overrightarrow{in}.\, \Subset^F_{\overrightarrow{in}, \overrightarrow{out}} \circ \langle\!\langle d \rangle\!\rangle^{\overrightarrow{in}, \overrightarrow{out}}$, for some choice of $\overrightarrow{out} \in \mathcal{I}_{\text{out}}$ and $F \in \mathcal{F}$.

Theorem 1. *A locality-preserving translation* $\mathbb{A} \to \mathbb{B}$ *is a sound translation.*

This theorem is proved by inductively transforming a high-level proof in $\mathbb{A}$ to the corresponding proof in $\mathbb{B}$, preserving the structure. Application preservation and crust inclusion allow us to transform a high-level frame into a low-level frame, and axiom correctness allows us to soundly replace the high-level commands with their implementations. The remaining proof rules transform naturally. If we chose to include the conjunction rule in our proof system, then we would need to additionally verify that our representation functions preserve conjunction and also that the crust predicate $\exists \overrightarrow{in}.\, \Cap{F}{\overrightarrow{in},\overrightarrow{out}}$ is precise.

5.1 Module Translation: $\mathbb{T} \to \mathbb{H} + \mathbb{L}$

We study a list-based implementation of the tree module which uses a combination of the heap and list modules given in §3. We shall see that this implementation provides a locality-preserving translation of our abstract tree module. To define a locality-preserving translation we need to give a representation function, a crust predicate and a substitutive implementation function for the translation.

The representation functions for trees and tree contexts are given below. As we have seen, each node of the tree is represented by a list of addresses of the node's children and a memory block of two fields that record the addresses of the parent node and child list. The *in* part of the interface, $l \in (\mathbb{N}^+)^*$, is a list of the addresses of the top-level nodes of the subtree. The *out* part of the interface, $u \in \mathbb{N}^+$, is the address of the subtree's parent node. We use the notation $x \mapsto y,z$ to mean $x \mapsto y * x+1 \mapsto z$ and also write $\mathtt{E}_1 \doteq \mathtt{E}_2$ to mean $\mathrm{emp} \wedge (\mathtt{E}_1 = \mathtt{E}_2)$. We also abuse notation slightly, freely combining heaps and list stores with $*$.

$$
\begin{aligned}
\langle\!\langle \varnothing \rangle\!\rangle^{\varepsilon,u} &::= \mathrm{emp} \\
\langle\!\langle n[t] \rangle\!\rangle^{n,u} &::= \exists i, l.\, n \mapsto u,i * i \Mapsto [\, l \,] * \langle\!\langle t \rangle\!\rangle^{l,n} \\
\langle\!\langle t_1 \otimes t_2 \rangle\!\rangle^{l,u} &::= \exists l_1, l_2.\, (l \doteq l_1 + l_2) * \langle\!\langle t_1 \rangle\!\rangle^{l_1,u} * \langle\!\langle t_2 \rangle\!\rangle^{l_2,u} \\
\\
\langle\!\langle - \rangle\!\rangle^{l,u}_{l',u'} &::= (l \doteq l') * (u \doteq u') \\
\langle\!\langle n[c] \rangle\!\rangle^{n,u}_{I'} &::= \exists i, l.\, n \mapsto u,i * i \Mapsto [\, l \,] * \langle\!\langle c \rangle\!\rangle^{l,n}_{I'} \\
\langle\!\langle t \otimes c \rangle\!\rangle^{l,u}_{I'} &::= \exists l_1, l_2.\, (l \doteq l_1 + l_2) * \langle\!\langle t \rangle\!\rangle^{l_1,u} * \langle\!\langle c \rangle\!\rangle^{l_2,u}_{I'} \\
\langle\!\langle c \otimes t \rangle\!\rangle^{l,u}_{I'} &::= \exists l_1, l_2.\, (l \doteq l_1 + l_2) * \langle\!\langle c \rangle\!\rangle^{l_1,u}_{I'} * \langle\!\langle t \rangle\!\rangle^{l_2,u}
\end{aligned}
$$

The crust, $\Cap{F}{I}$, parameterised by interface $I = l, u$ and free logical variables $F = (l_1, l_2, u')$, is defined as follows:

$$
\Cap{l_1,l_2,u'}{l,u} ::= \exists i.\, u \mapsto u', i * i \Mapsto [\, l_1 + l + l_2 \,] * \left(\prod_{n \in l_1 + l_2}^{*} n \mapsto u \right)
$$

This crust predicate captures the shaded part of the tree shown in Fig. 3(c) which includes the parent node u of the subtree and the list of u's children, including the top level nodes l of the subtree. These are needed by the implementations of commands that lookup siblings or parents, or delete a subtree. The crust also

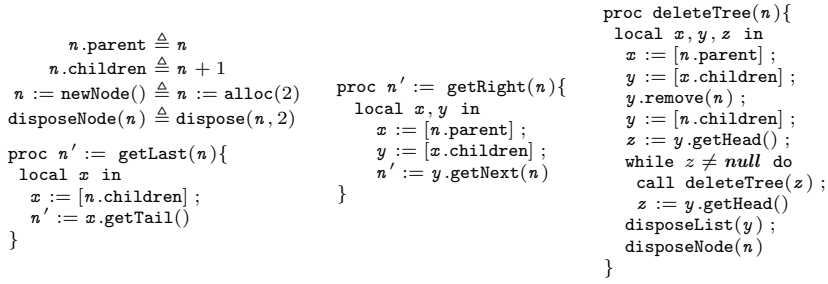

Fig. 4. Selected procedures for the list-based implementation

includes the other sibling nodes as these are needed by the implementation of the node insertion command.

A selection of the procedures that constitute the substitutive implementation function is given in Fig. 4.

Theorem 2. *The representation function, crust predicate and substitutive implementation given above constitute a locality-preserving translation.*

5.2 Module Translation: $\mathbb{H} + \mathbb{H} \to \mathbb{H}$

Another example of a locality-preserving translation is given by implementing a pair of heap modules $\mathbb{H}+\mathbb{H}$ in a single heap $\mathbb{H}$, by simply treating the two heaps as disjoint portions of the same heap.

The representation function is the same both for states and for contexts: $\langle\!\langle (h_1, h_2) \rangle\!\rangle = \{h_1\} * \{h_2\}$. The interface set is trivial (just a single-element set). The crust parameter set is also trivial, and the crust predicate is simply emp. The substitutive implementation $[\![\mathbb{C}]\!]$ is defined to be the detagging of $\mathbb{C}$: that is, heap commands from both abstract modules are substituted with the corresponding command from the single abstract module. For example:

$$[\![n := \mathtt{alloc}_1(E)]\!] = n := \mathtt{alloc}(E) = [\![n := \mathtt{alloc}_2(E)]\!]$$

Theorem 3. *The representation function, crust predicate and substitutive implementation given above constitute a locality-preserving translation.*

(Note, the representation function in this case does not preserve conjunction.)

6 Locality-breaking Translations

There is not always a close correspondence between locality in an abstract module and locality in its implementation. For example, consider an implementation of our list module that represents each list as a singly-linked list in the heap. In the abstract module, the footprint of removing a specific element from a list

is just the element of the list. In the implementation however, the list is traversed from its head to reach the element, which is then deleted by modifying the pointer of its predecessor. The footprint is therefore the list fragment from the head of the list to the element, significantly more than the single list node holding the value to be removed. While we could treat this additional footprint as crust, in this case it seems appropriate to abandon the preservation of locality and instead give a locality-breaking translation that provides a fiction of locality.

Consider a translation from abstract module $\mathbb{A}$ to $\mathbb{B}$. With the exception of the frame rule and axioms, the proof rules for $\mathbb{A}$ can be mapped to the corresponding proof rules of $\mathbb{B}$: that is, from the translated premises we can directly deduce the translated conclusion. To deal with the frame rule, we remove it from proofs in $\mathbb{A}$ by 'pushing' applications of the frame rule to the leaves of the proof tree. In this way, we can transform any local proof to a non-local proof.

Lemma 1 (Frame-free Derivations). *Let $\mathbb{A}$ be an abstract module. If there is a 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:*

$$\dfrac{\dfrac{}{\Gamma \vdash \{p\}\ \mathbb{C}\ \{q\}}\ (\dagger)}{\Gamma \vdash \{f \circ p\}\ \mathbb{C}\ \{f \circ q\}} \qquad \dfrac{\dfrac{\vdots}{\Gamma \vdash \{p\}\ \mathbb{C}\ \{q\}}}{\Gamma \vdash \{(\mathbf{I}_{\mathbb{A}} \times \sigma) \circ p\}\ \mathbb{C}\ \{(\mathbf{I}_{\mathbb{A}} \times \sigma) \circ q\}}$$

where (†) *is either* Axiom, Skip *or* Assgn.

By transforming a high-level proof of $\vdash_{\mathbb{A}} \{p\}\ \mathbb{C}\ \{q\}$ in this way, we can establish $\vdash_{\mathbb{B}} \{[\![p]\!]\}\ [\![\mathbb{C}]\!]\ \{[\![q]\!]\}$ provided that we can prove that the implementation of each command of $\Phi_{\mathbb{A}}$ satisfies the translation of each of its axioms under every frame. (We can reduce considerations to *singleton* frames by considering any given frame as a disjunction of singletons and applying the Disj rule.)

Definition 16 (Locality-breaking Translation). *A* locality-breaking translation $\mathbb{A} \to \mathbb{B}$ *is a module translation such that, for all $c \in \mathcal{C}_{\mathbb{A}}$ and $(p, \varphi, q) \in \mathrm{Ax}_{\mathbb{A}}$, the judgment $\vdash_{\mathbb{B}} \{[\![\{c\} \circ p]\!]\}\ [\![\varphi]\!]\ \{[\![\{c\} \circ q]\!]\}$ holds.*

Theorem 4. *A locality-breaking translation is a sound translation.*

If we include the conjunction rule, then we must verify that every singleton context predicate is precise (i.e. the context algebra must be left-cancellative). Note that, whilst there is less to prove when working with locality-breaking translations than with locality-preserving translations, the actual proofs may be more difficult as we have to show that the axioms are preserved in *every* context. Our examples show that the two approaches are suited to different circumstances.

6.1 Module Translation: $\mathbb{L} \to \mathbb{H}$

We provide a locality-breaking translation $\mathbb{L} \to \mathbb{H}$, which implements abstract lists with singly-linked lists in the heap.

```
x.value ≜ x
x.next ≜ x + 1
x := newNode() ≜ x := alloc(2)
disposeNode(x) ≜ dispose(x, 2)

proc i.remove(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)
}

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

Fig. 5. Selected procedures for the linked-list implementation

The state translation from list stores to heaps is defined inductively by:

$$[\![\emptyset]\!] ::= \text{emp} \qquad [\![i \Mapsto l * ls]\!] ::= \mathbf{False}$$
$$[\![i \Mapsto [\,l\,] * ls]\!] ::= \exists x.\, i \mapsto x * \langle\!\langle l \rangle\!\rangle^{(x,\mathbf{null})} * [\![ls]\!]$$

where

$$\langle\!\langle \varepsilon \rangle\!\rangle^{(x,y)} ::= (x \doteq y) \qquad \langle\!\langle v \rangle\!\rangle^{(x,y)} ::= x \mapsto v,y$$
$$\langle\!\langle l + l' \rangle\!\rangle^{(x,y)} ::= \exists z.\, \langle\!\langle l \rangle\!\rangle^{(x,z)} * \langle\!\langle l' \rangle\!\rangle^{(z,y)}$$

Note that not all list stores are realised by heaps: only ones in which every list is complete. The intuition behind this is that partial lists are purely abstract notions that provide a useful means to our ultimate end, namely reasoning about complete lists. The abstract module itself does not provide operations for creating or destroying partial lists, and so we would not expect to give specifications for complete programs that concern partial lists. A selection of the procedures that constitute the substitutive implementation function is given in Fig. 5.

Theorem 5. *The state translation and substitutive implementation given above constitute a locality-breaking translation.*

7 Conclusions

We have shown how to refine module specifications given by abstract local reasoning into correct implementations. We have identified two general approaches for proving correctness: 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. Meanwhile, locality-breaking translations establish a fiction of locality, by justifying abstract locality even though this locality is not matched by the implementation.

This paper has focused on refinement for abstract local reasoning in the sequential setting. With Dodds, Parkinson and Vafeiadis, Dinsdale-Young and

Gardner have introduced concurrent abstract predicates [6] as a technique for verifying correct implementations of concurrent modules. They achieve local reasoning and disjoint concurrency at the abstract level, by abstracting from a low-level resource model with fine-grained permissions. Our next challenge is to extend our refinement techniques to the setting of disjoint concurrency.

Acknowledgments: Gardner acknowledges support of a Microsoft/RAEng Senior Research Fellowship. Dinsdale-Young and Wheelhouse acknowledge 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.

References

1. R. Bornat, C. Calcagno, and H. Yang. Variables as resource in separation logic. In *Proceedings of MFPS XXI*, volume 155 of *ENTCS*, pages 247–276, Amsterdam, The Netherlands, 2006. Elsevier.
2. C. Calcagno, P. Gardner, and U. Zarfaty. Context logic and tree update. In *POPL '05*, volume 40 of *SIGPLAN Not.*, pages 271–282, New York, NY, USA, 2005. ACM.
3. C. Calcagno, P. Gardner, and U. Zarfaty. Local reasoning about data update. *Electron. Notes Theor. Comput. Sci.*, 172:133–175, 2007.
4. C. Calcagno, P. W. O'Hearn, and H. Yang. Local action and abstract separation logic. In *LICS '07*, pages 366–378, Washington, DC, USA, 2007. IEEE Computer Society.
5. W. DeRoever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and Their Comparison.* Cambridge University Press, Cambridge, UK, 1999.
6. T. Dinsdale-Young, M. Dodds, P. Gardner, M. Parkinson, and V. Vafeiadis. Concurrent abstract predicates. In *ECOOP '10*, LNCS, Berlin/Heidelberg, Germany, 2010. Springer.
7. T. Dinsdale-Young, P. Gardner, and M. Wheelhouse. Abstract local reasoning. Technical report, Imperial College London, London, UK, 2010. `http://www.doc.ic.ac.uk/~td202/papers/alrfull.pdf`.
8. P. A. Gardner, G. D. Smith, M. J. Wheelhouse, and U. D. Zarfaty. Local Hoare reasoning about DOM. In *PODS '08*, pages 261–270, New York, NY, USA, 2008. ACM.
9. J. He, C. A. R. Hoare, and J. W. Sanders. Data refinement refined. In *ESOP '86*, volume 213 of *LNCS*, pages 187–196, Berlin/Heidelberg, Germany, 1986. Springer.
10. C. A. R. Hoare. Proof of correctness of data representations. *Acta Inf.*, 1(4):271–281, 1972.
11. I. Mijajlović, N. Torp-Smith, and P. W. O'Hearn. Refinement and separation contexts. In *FSTTCS '04*, pages 421–433, Berlin/Heidelberg, Germany, 2004. Springer.
12. P. W. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *CSL '01*, volume 2142 of *LNCS*, pages 1–19, Berlin/Heidelberg, Germany, 2001. Springer.
13. M. Parkinson and G. Bierman. Separation logic and abstraction. In *POPL '05*, volume 40 of *SIGPLAN Not.*, pages 247–258, New York, NY, USA, 2005. ACM.