Specification patterns for reasoning about recursion through the store

Nathaniel Charlton, Bernhard Reus*

Department of Informatics, University of Sussex, Falmer, Brighton, United Kingdom

Abstract

Higher-order store means that code can be stored on the mutable heap that programs manipulate, and is the basis of flexible software that can be changed or reconfigured at runtime. Specifying such programs is challenging because higher-order store allows *recursion through the store*, where new (mutual) recursions between code are set up on the fly. This paper presents a series of formal specification patterns that capture increasingly complex uses of recursion through the store. To express the necessary specifications we extend the separation logic for higher-order store given by Schwinghammer *et al.* (CSL, 2009), adding parameter passing, and certain recursively defined families of assertions. We give proof rules for our extended logic and show their soundness. Finally, we apply our specification patterns and rules to an example program that exploits many of the possibilities offered by higher-order store; this is the first larger case study conducted with logical techniques based on work by Schwinghammer *et al.* (CSL, 2009), and shows that they are practical.

Keywords: Hoare logic, higher-order store, separation logic

1. Introduction and motivation

Popular "classic" languages like ML, Java and C provide facilities for manipulating code stored on the heap at runtime. With ML one can store newly generated function values in heap cells; with Java one can load new classes at runtime and create objects of those classes on the heap. Even for C, where the code of the program is usually assumed to be immutable, programs can dynamically load and unload libraries at runtime, and use function pointers to invoke their code. Heaps that contain code in this way have been termed *higher-order store*.

*Corresponding author. Tel. +44 (0)1273 678195. Fax +44 (0)1273 877873.
Dept. of Informatics, University of Sussex, Falmer, Brighton, BN1 9QJ, United Kingdom
Email addresses: billiejoecharlton@gmail.com (Nathaniel Charlton), bernhard@sussex.ac.uk (Bernhard Reus)

This important language feature is the basis of flexible software systems that can be changed or reconfigured at runtime. For example, the module mechanism of the Linux kernel allows one to load, unload and update code which extends the functionality of the kernel, without rebooting the system [1]. Examples of modules include drivers for hardware devices and filesystems, and executable interpreters that provide support for running new kinds of executables; by updating function pointers in the "syscall table", modules can at run time intercept any system call that the kernel provides. In [2, 3] bugfixing and upgrading C programs without restarting them is discussed; for instance, a version of the OpenSSH server is built that can update itself while running when a new version becomes available, without disconnecting existing users.

Obtaining logics, and therefore verification methods, for such programs has been very challenging however, due to the complexity of higher-order heaps (see for example the discussion in [4]). When using denotational semantics, the denotation of such a heap is a mixed-variant recursively defined domain. The recursive nature of the heap complicates matters, because in addition to loading, updating and deallocating code, programs may "tie knots in the store" [5], i.e. create new recursions on the fly; this is known as *recursion through the store*. In fact, this knot-tying is happening whenever code on the heap is used in a recursive way, such as in the Visitor pattern [6] which involves a mutual recursion between the visitor's methods and the visited structure's methods.

Existing work [7] and [8] has discussed reasoning about recursion through the store, but only using the simplest recursion pattern needed to implement the factorial function by recursion through the store (see our pattern in Section 4 with a fixed code pointer).

To enable logical reasoning about software that uses higher-order store, the contributions of this paper are as follows.

- We extend the logic of [9], adding parameter passing, a first-order treatment of finite sets, inductively defined predicates and certain recursively defined families of assertions (Section 3). Proof rules for these extensions are given and shown to be sound (Section 5).

- We present and classify patterns of formal specification for programs that recurse through the store, using recursive assertions and nested triples (Section 4). ([9], on which we build, considered only a very simple form for specifications.)

- We state a generic "master pattern" including combinations of inductive and recursive predicate definitions that covers all patterns we identified in this paper, and argue that the fixpoints needed to give semantics to such specifications always exist (Section 4.1).

- We apply the specification and proof techniques we developed to an example program which exploits many of the possibilities offered by higher-order store (Section 6). This is the first larger case study conducted with logical techniques based on [9], and shows that they are practical. We

give some details of the proofs, and point out that we have developed a verification tool [10] to support these.

The next section will give an example program which uses higher-order store and recursion through the store. Before presenting this example program, we identify some of the possibilities offered by higher-order store.

Remark 1.1. Some possibilities offered by higher-order store.

1. **Flexible composition of components:** Software components can be stored on the heap and connected together using code pointers (which in general leads to recursion through the store). By updating these pointers the main program can, at runtime, restructure the way the components are used. Further, if the main program provides a directory listing the components present, then components can also discover each other dynamically.
2. **Dynamic, on-demand loading of code:** Software components can be implemented as separate modules which are then dynamically loaded by the main program. Thus code can be loaded lazily, i.e. only when it is actually needed. Further, if the main program provides the components with an interface to the loader, then components can take care of dynamically loading other components on which they depend.
3. **Self-updating code:** Software components can be programmed to update themselves, for example periodically checking for newer versions of themselves on disk or over a network, and replacing themselves when these become available. This feature may be particularly useful in cases where stopping the system to perform upgrades is unacceptable.
4. **Specialisation of code at runtime:** Specialised implementations of functions can be created at runtime and integrated fully into the system. E.g. for efficiency purposes, a component may create an implementation of a function which is specialised to the invocation cases which are occurring frequently (see [11]).
5. **Unloading of code which is no longer needed:** The main program can unload components which are no longer in use. If an interface for unloading is provided, then components can decide to unload themselves or each other.

The remainder of this article is structured as follows. Section 2 explains our running example program. This program will be revisited in Section 6 where it is sketched how one can show correctness of the sample program using our logic for higher-order store. Section 3 presents the programming language and assertion language. In Section 4 we will describe various patterns of specifications for programs that recurse through the store and present a generic pattern for which we show existence. The proof rules and their soundness are discussed in Section 5 building on existing work of [9, 12]. Section 7 outlines further research.

This article is an extended version of our conference paper [13] and contains additional details on inductive definitions, an extended example, two extra patterns, a more general "master pattern", and more details about proof rules and soundness proofs.

2. Our running example program

We now present an idealised but concrete example program which demonstrates in a simple way some of the possibilities offered by higher-order store and recursion through the store, of which it makes essential use. We will revisit this program in Section 6 and show that our logic for higher-order store is capable of reasoning about all these uses.

Our program performs evaluation of (binary) expressions represented as binary trees. A tree node is either an integer leaf or a binary fork annotated with a binary operator. The distinction is effected by labels: "opLbl", which is 0, for operators, and "leafLbl", which is 1, for leaves. For operations we will look at some typical examples like PLUS, but it is inherent to the approach that any (binary) operation can be interpreted. This flexibility is achieved by giving each fork node a pointer to the evaluation procedure to be used to evaluate the relevant operator. The referenced evaluation procedure "implements" the meaning of the labeled node. This flexibility goes beyond the classic "visitor" pattern, which only works for a predefined class of node types. Unary operators can be included as binary operators which ignore their second argument, and variables can be included by a distinguished operator VAR, where "VAR n z" represents the nth variable x_n and the expression z is ignored.

Importantly the code implementing the various evaluations of different tree nodes is *not fixed* by the main program; instead, each operator evaluation procedure is implemented as a *loadable module*, which can be loaded on demand and a pointer to which can be saved in the tree nodes. This results in the data structures shown in Fig. 1.

The main program. The code for the main part of our tree evaluator is given in Fig. 2. The $\mathsf{eval}\ [e](\vec{e})$ statement invokes the code stored in the heap at address e, with a vector of (value) parameters $\vec{e}$. The shorthand res (explained later) simulates reference parameters. The expression '$\lambda\vec{x}.C$' denotes an *unevaluated* procedure with body C, taking formal parameters $\vec{x}$, as a value; thus $[e] := \text{'}\lambda\vec{x}.C\text{'}$ is used to store procedures into the heap. As in ML, all variables in our language are immutable, so that once they are bound to a value, this value does not change. This property of the language lets us avoid side conditions on variables when studying frame rules. Our main program's code assumes the following to be in the heap:

1. The input: variable *tree* is a pointer to a binary tree as described above situated on the heap; *res* is a reference cell to store the result of the evaluation. Because such expression trees can include variables, an association list mapping those variables to values is globally available at address *assoclist*.
2. Module-loading infrastructure: consists of a linked list storing modules, pointed to by a constant pointer *modlist*, and two procedures pointed to by *searchMods* and *loader*. Calling $\mathsf{eval}\,[\mathit{searchMods}](\mathit{opID}, \mathsf{res}\ \mathit{codeaddr})$ searches the list of loaded modules for the one implementing operator *opID*, returning its address or null (0) if it is not present. Calling the loader via $\mathsf{eval}\ [\mathit{loader}](\mathit{opID}, \mathsf{res}\ \mathit{codeaddr})$ always *guarantees* a module

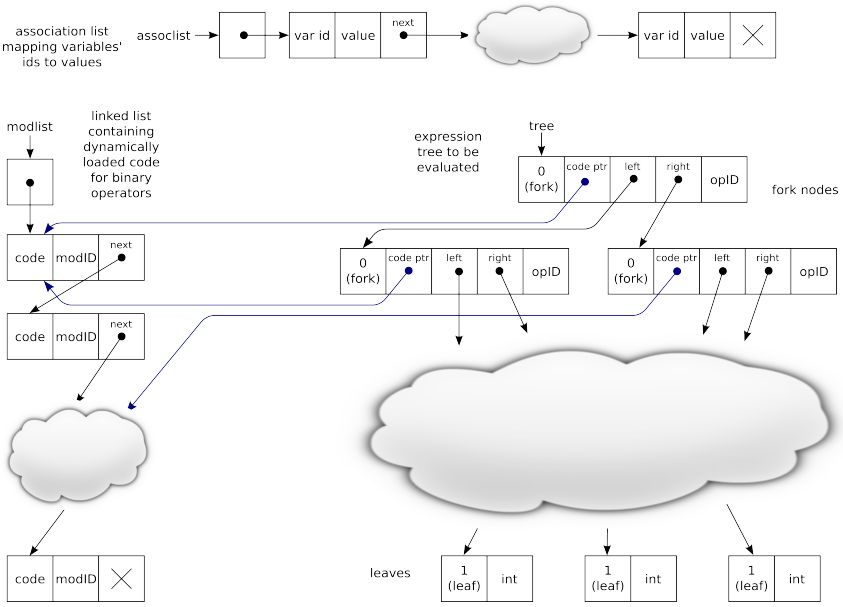

Figure 1: The data structures used by our example program.

implementing operator *opID* is loaded, loading it if necessary, and returning its address.[1]

3. A "tree visitor" procedure pointed to by *evalTree*, whose address is known to all modules and the main program. Note that this visitor does not contain the individual procedures for node types as in the standard pattern because we can directly store pointers to them within the nodes.

The main program first stores the procedure *evalTree* in the heap before calling it for the given input tree and result cell. The evaluation procedure will in turn call the procedures referenced in the tree nodes which may call the evaluation procedure back, thus giving rise to recursion through the store. For ease of presentation this code assumes that the tree and a suitable global *modlist* are already set up; we do not describe the initial construction of these data structures. We will, however, demonstrate that further module loading can be done once the evaluator is already in action, namely from one of the procedures called by *evalTree*.

Some illustrative modules. Independently of the main program we can write the loadable modules starting with the basic ones for the evaluation of

[1] We could give the code for *searchMods* but we won't for brevity. We could also give the code for *loader* in terms of a more primitive loading operation, separating out the list manipulation from the code loading, but a built-in loading operator would still be necessary.

```
// constant offsets
const CodePtrO = 1
const LeftO = 2
const RightO = 3
const OpIDO = 4
const ValO = 1


// constant labels for trees
const leafLabel = 1
const opLabel = 0
```

```
[evalTree] :=
 ‘λ tree, resaddr.
   let kind = [tree] in
    if kind = leafLabel then
     let val = [tree + ValO] in  [resaddr] := val
    else
     let codePtr = [tree + CodePtrO] in
      eval [codePtr](tree, resaddr)
 ’ ;
eval [evalTree](tree, res res) ;

let disposeTree = new 0 in
 [disposeTree] :=
  ‘λ tree.
    let kind = [tree] in
     if kind = leafLabel then
      free tree ;
      free tree + ValO
     else
      let left = [tree + leftO] in
      let right = [tree + rightO] in
       free tree ;
       free tree + CodePtrO ;
       free tree + LeftO ;
       free tree + RightO ;
       free tree + OpIDO ;
       eval [disposeTree](left) ;
       eval [disposeTree](right) ’ ;
 eval [disposeTree](tree)
```

Figure 2: Code for the “main program” part of our running example.

nodes that are labeled VAR, PLUS, TIMES etc. The VAR module evaluates its left subtree to an integer n, and then looks up the value of x_n, the variable with ID n, from the association list (the right subtree is ignored). Fig. 3 contains an implementation of PLUS. Note how this implementation calls back *evalTree* which in turn makes further calls to modules (either for PLUS again or for other operators): this is mutual recursion through the store.

As well as implementing arithmetic operators, the module mechanism can

```
PLUS: ‘λ tree, resaddr.
  let left = [tree + LeftO] in
  let right = [tree + RightO] in
    eval [evalTree](left, res leftVal) ;
    eval [evalTree](right, res rightVal) ;
    [resaddr] := leftVal + rightVal’

WHILE: ‘λ tree, resaddr.
  let left = [tree + LeftO] in
  let right = [tree + RightO] in
  let b = new 0 in
    eval [evalTree](left, b) ;
    while [b] do
      (eval [evalTree](right, resaddr) ;
      eval [evalTree](left, b)) ;
    free b’

OSCILLATE : ‘λ tree, resaddr.
  let left = [tree + LeftO] in
    eval [evalTree](left, resaddr) ;
    let selfCodeptr = [tree + CodePtrO]
      in
      let oldCode = [selfCodeptr] in
      [selfCodeptr] :=
        ‘λ tree, resaddr.
          let right = [tree + RightO] in
            eval [evalTree](right, resaddr) ;
            let selfCodeptr =
              [tree + CodePtrO] in
              [selfCodeptr] := oldCode ’ ’

LOAD_OVERWRITE : ‘λ tree, resaddr.
  let opcode = [tree + OpIDO] in
    eval [loader](opcode, res procptraddr) ;
    [tree + CodePtrO] := procptraddr
    eval [evalTree](tree, resaddr)’

COPRIME_GCD : ‘λ tree, resaddr.
  eval [loader](GCD, res gcdCodePtr) ;
  eval [loader](EQUAL, res eqCodePtr) ;
  let left = [tree + LeftO] in
  let right = [tree + RightO] in
  let newfork =
    new opLbl, gcdCodePtr, left, right, 0 in
  let newoneleaf = new leafLbl, 1 in
    [tree] := eqCodePtr ;
    [tree + LeftO] := newfork ;
    [tree + RightO] := newoneleaf ;
    eval [evalTree](tree, resaddr)’

COPRIME_choice : ‘λ tree, resaddr.
  // First find own address in memory
  eval [loader](COPRIME, res selfPtr) ;
  // Find out whether the GCD
  // operator is already loaded
  eval [searchMods](GCD, res gcdPtr) ;
  (if gcdPtr = null then
    // Overwrite our own code
    // with the version that uses LCM
    [selfPtr] := COPRIME_LCM
  else // Overwrite our own code
    // with the version that uses GCD
    [selfPtr] := COPRIME_GCD) ;
  eval [evalTree](tree, resaddr)’
```

Figure 3: Code for some modules demonstrating various uses of higher-order store.

```
BINOM: ‘λ tree, resaddr.
 let Fact = new ‘λ n, resaddr. FACT’ in
 let left = [tree + LeftO] in
 let right = [tree + RightO] in
 let leftkind = [left] in
  if leftkind = leafLabel then // if we’re specialising
   let n = [left + ValO] in
    eval [Fact](n, res nfact) ;
    // Create a new list node with the specialised code in it
    let l = [modlist] in
    let newnode = new
     ‘λ tree, resaddr.
       let Fact = new ‘λ n, resaddr . FACT’ in
       let right = [tree + RightO] in
         eval [evalTree](right, res k) ;
         eval [Fact](k, res kfact) ;
         eval [Fact](n − k, res nminuskfact) ;
         [resaddr] :=
           nfact/(kfact ∗ nminuskfact) ;
         free Fact
     ’,   0, l
   in
     free Fact ; [modlist] := newnode ;
     // Now change the code pointer for the
     // current tree node and recurse
     [tree + CodePtrO] := newnode ;
     eval [evalTree](tree, resaddr)
  else // we’re doing a normal binomial
  // calculation, not specialising
   eval [evalTree](left, res n) ;
   eval [evalTree](right, res k) ;
   eval [Fact](n, res nfact) ;
   eval [Fact](k, res kfact) ;
   eval [Fact](n − k, res nminuskfact) ;
   [resaddr] := nfact/(kfact ∗ nminuskfact) ;
   free Fact’
```

Figure 4: Code for the BINOM module.

be used to extend the program in more dramatic ways. We can implement an operator ASSIGN, so that expression ASSIGN E_1 E_2 *updates* the association list, giving variable x_{E_1} the new value E_2, and returns, say, the variable's new value. Then we can turn our expression evaluator into an interpreter for a programming language: we can add modules implementing the usual control constructs such as sequential composition, alternation and repetition. Fig. 3 gives the implementation for WHILE. We emphasise that the WHILE operator can only be implemented because the code for each operator decides how often and when to evaluate the subexpressions; if the main program were in charge of the tree traversal (i.e. a tree fold was being used), WHILE could not be written.

Further modules in Fig. 3 illustrate more complex uses of higher-order store. Consider the operator COPRIME which tests whether its two arguments are coprime. $\text{COPRIME}_{\text{GCD}}$ implements this operator, but depends on another module GCD implementing the greatest common divisor operator. When it runs, $\text{COPRIME}_{\text{GCD}}$ uses the *loader* procedure provided by the main program to make sure that the GCD and equality code is in memory, loading them if necessary; the expression tree is then patched appropriately, and *evalTree* is called recursively on the new subtree. One can similarly program an implementation $\text{COPRIME}_{\text{LCM}}$ which depends on the least common multiple operator LCM. The module $\text{COPRIME}_{\text{choice}}$ takes things further: when first run, it checks the list of currently loaded code to discover whether the GCD module is already present. If so, the GCD-based implementation is preferred, and $\text{COPRIME}_{\text{choice}}$ *updates itself* in-place with the GCD-based implementation; if not, the LCM-based implementation is preferred and the code ovewrites itself with $\text{COPRIME}_{\text{LCM}}$.

The LOAD_OVERWRITE procedure first loads the code for the tree node's *opID* into the module list, then updates the node's code pointer before calling that to evaluate the tree with the freshly loaded procedure. Note that next time the same fork node is visited the newly loaded procedure is executed straight away and no more loading occurs. Though we do not do so here, unloading of code could also be added: a module in the code list not pointed to by any tree node can be safely disposed.

The operator OSCILLATE chooses to evaluate the left subtree and returns its result. But it also updates itself with a version that, when evaluated, picks the right subtree for evaluation and then updates back to the original version. In this case the code in the module list itself is updated and thus *all* tree references pointing to it from the tree are affected by the update.

We finish by examining an even more complicated module BINOM (Fig. 4) that implements the binomial coefficient operator $\binom{n}{k} := n!/k!(n-k)!$ (using some code *FACT* that calculates factorials).

This implementation demonstrates the *specialisation of code at runtime*: it detects cases in which the left subtree (corresponding to n) is an integer literal. In such cases the implementation calculates $n!$ and generates on the fly an optimised implementation which reuses the value on all future invocations. The specialised implementation is added to the list of loaded modules and pays off each subsequent time the $\binom{n}{k}$ subexpression is evaluated (which may be many

$$e ::= \; 0 \mid 1 \mid -1 \mid \ldots \mid e_1 + e_2 \mid \ldots \mid x \mid \text{`}\lambda\vec{x}.C\text{'} \qquad \Sigma ::= \; x \mid \{e\} \mid \emptyset \mid \Sigma_1 \cup \Sigma_2$$

$$\begin{aligned} C ::= \;& [e_1] := e_2 \mid \mathsf{let}\ y = [e]\ \mathsf{in}\ C \mid \mathsf{eval}\ [e](\vec{e}) \mid \mathsf{let}\ x = \mathsf{new}\ \vec{e}\ \mathsf{in}\ C \\ & \mid \mathsf{free}\ e \mid \mathsf{skip} \mid C_1; C_2 \mid \mathsf{if}\ e_1 = e_2\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \end{aligned}$$

$$\begin{aligned} P ::= \;& \mathrm{True} \mid \mathrm{False} \mid P_1 \vee P_2 \mid P_1 \wedge P_2 \mid P_1 \Rightarrow P_2 \mid \forall x.P \mid \exists x.P \\ & \mid e_1 = e_2 \mid e_1 \leq e_2 \\ & \mid e_1 \mapsto e_2 \mid \mathsf{emp} \mid P_1 \star P_2 \\ & \mid \mathfrak{P}(\vec{e}) \mid \{P\}\, e(\vec{e})\, \{Q\} \mid P \otimes Q \mid \Sigma_1 \subseteq \Sigma_2 \end{aligned}$$

$$\mathfrak{P} ::= \; \mathrm{R} \mid \mu^k\, \mathrm{R}_1(\vec{x}_1), \ldots, \mathrm{R}_n(\vec{x}_n)\ .\ P_1, \ldots, P_n \mid \mathfrak{I}(\vec{\mathfrak{P}})$$

$$\mathfrak{I} ::= \; \mu^k_{ind} \left\langle \vec{\mathrm{R}} \right\rangle \underline{\mathrm{I}}_1(\vec{x}_1), \ldots, \underline{\mathrm{I}}_n(\vec{x}_n).\ P_1, \ldots, P_n$$

Figure 5: Syntax of expressions, commands and assertions.

times if it is inside a while loop, for instance). One sees in the code how the specialisation is achieved: the value *nfact* is calculated outside the scope of the innermost quoted code, but then used within it. We assume that all the tokens for operators appearing in inputs will be different from 0, so we can use 0 in the `modID` field (see Figure 1) when adding dynamically generated code to the linked list.

One can similarly imagine using code generation for exponentiation operations where the power is a literal; the generated code will be simply a sequence of multiplications. (This is essentially the "staged power" example from [14]).

3. The programming and assertion languages

We now introduce the programming and assertion languages we work with.

3.1. The programming language

We use a simple imperative programming language extended with operations for stored procedures and heap manipulation as described and used in Section 2. The syntax of the language is shown in Fig. 5, where $\vec{x}$ (resp. $\vec{e}$) represents a vector of distinct variables (resp. a vector of expressions). This language extends that in [9] by providing for the passing of value parameters. For convenience we employ two abbreviations: we allow ourselves a looping construct $\mathsf{while}\ [e]\ \mathsf{do}\ C$, which can be expressed with recursion through the store, and we write $\mathsf{eval}\ [e](\vec{e}, \mathsf{res}\ v)\ ;\ C$ as shorthand for

$$\mathsf{let}\ vaddr = \mathsf{new}\ 0\ \mathsf{in}\ (\mathsf{eval}\ [e](\vec{e}, vaddr)\ ;\ \mathsf{let}\ v = [vaddr]\ \mathsf{in}\ C\ ;\ \mathsf{free}\ vaddr)$$

The expressions in the language are integer expressions, variables, and the quote expression '$\lambda\vec{x}.C$' for representing an (unevaluated) procedure C taking formal parameters $\vec{x}$. The quotes are useful brackets that help to actually determine where some stored procedure ends and the top-level code continues. Note that $fv(\text{`}\lambda\vec{x}.C\text{'}) := fv(C) - \vec{x}$.

The integer or code value denoted by expression e_1 can be stored in a heap cell e_0 using $[e_0]{:=}e_1$, and this stored value can later be looked up and bound to the (immutable) variable y by $\mathsf{let}\ y = [e_0]\ \mathsf{in}\ D$. In case the value stored in cell e_0 is code '$\lambda\vec{x}.C$', we can run (or "evaluate") this code with actual parameters $\vec{e}$ by executing $\mathsf{eval}\ [e_0](\vec{e})$. Our language also provides constructs for allocating and disposing heap cells such as e_0 above.

3.2. The assertion language

The assertion language, shown in Fig. 5, follows [9], adding finite sets, more general recursive definitions and inductive predicates, and with some changes to accommodate parameter passing. Each assertion describes a property of states, which consist of an (immutable) environment and a mutable heap. The language is based on first-order intuitionistic logic with several further extensions:

Separation logic: Our language includes the $\star$, emp and $\mapsto$ connectives of separation logic [15]. We use some abbreviations to improve readability: $e \in \Sigma := \{e\} \subseteq \Sigma$, $e \mapsto \_ := \exists x.\, e \mapsto x$ and $e \mapsto P[\cdot] := \exists x.\, e \mapsto x \wedge P[x]$ where $P[\cdot]$ is an assertion with an expression hole, such as $\{Q\}\cdot\{R\}$, $\cdot = e$ or $\cdot \leq e$. Additionally we have $e \mapsto e_0, \ldots, e_n := e \mapsto e_0 \star \cdots \star (e{+}n) \mapsto e_n$.

Nested triples: Triples are assertions, so they can appear in pre- and post-conditions of triples. This *nested* use of triples is crucial because it allows one to specify stored code behaviourally, i.e. in terms of properties that it satisfies and *not* by specifying the intensional properties of the code as e.g. in [16, 8] . The triple $\{P\}\, e(\vec{e})\, \{Q\}$ means that e denotes code satisfying $\{P\}\cdot\{Q\}$ when invoked with parameters $\vec{e}$. For code that does not expect any parameters, $\vec{e}$ will have length zero and we write simply $\{P\}\, e\, \{Q\}$.

Invariant extension: Intuitively, the invariant extension $P \otimes Q$ denotes a modification of P where the pre- and post-conditions of all triples inside P (at all nesting depths) are $\star$-extended with Q. The operator $\otimes$ is from [17, 9] and is not symmetric.

Inductively defined predicates: The syntactic category $\mathfrak{I}$ is for parametric, (mutually) inductively defined predicates. These are written in the form

$$\mu^k_{ind} \left\langle \vec{\mathrm{R}} \right\rangle \underline{\mathrm{I}}_1(\vec{x}_1), \ldots, \underline{\mathrm{I}}_n(\vec{x}_n).\ P_1, \ldots, P_n$$

which creates n inductively defined predicates and then projects out the kth one. Each P_i may contain free variables from parameters $\vec{x_i}$, free predicate variables $\vec{\mathrm{R}}$ for predicate parameters, and predicate variables for other (mutually inductively defined) predicates $\vec{\underline{\mathrm{I}}}$. To ensure that the

$$
\begin{array}{l}
\text{lseg} ::= \\
\quad \mu^1_{ind}\ \underline{\mathrm{I}}(x, y, \sigma). \\
\quad\quad \left(\begin{array}{l} \quad x = y \wedge \sigma = \emptyset \wedge \mathsf{emp} \\ \vee\ \exists nxt, \sigma' \,.\, x \mapsto \_, \_, nxt \star \underline{\mathrm{I}}(nxt, y, \sigma') \wedge nxt \neq x \wedge \sigma = \sigma' \cup \{x\} \end{array} \right)
\end{array}
$$

$$
\begin{array}{l}
\text{tree} ::= \\
\quad \mu^1_{ind}\ \underline{\mathrm{I}}(t, \tau). \\
\quad\quad \left(\begin{array}{l} \quad t \mapsto \text{leafLbl}, \_ \wedge \tau = \emptyset \\ \vee\ \exists codePtr, left, right, \tau', \tau''. \\ \qquad t \mapsto \text{opLbl}, codePtr, left, right, \_ \star \underline{\mathrm{I}}(left, \tau') \star \underline{\mathrm{I}}(right, \tau'') \\ \quad \wedge\ \tau = \{codePtr\} \cup \tau' \cup \tau'' \end{array} \right)
\end{array}
$$

$$
\begin{array}{l}
\text{tree}_{\text{fork}}(t, \tau) ::= \\
\quad \exists codePtr, left, right, \tau', \tau''. \\
\qquad t \mapsto \text{opLbl}, codePtr, left, right, \_ \star \text{tree}(left, \tau') \star \text{tree}(right, \tau'') \\
\quad \wedge\ \tau = \{codePtr\} \cup \tau' \cup \tau''
\end{array}
$$

Figure 6: Inductively defined predicates used to specify and prove our example program.

definition gives rise to an inductively defined predicate a sufficient (but not necessary) condition on the syntax of inductive definitions is that the predicates $\underline{\mathrm{I}}_i$ do not appear on the left hand side of implication, inside universal quantification or nested triples. Note that this is not too strong a restriction as one can define recursive predicates and use them as actual parameters for $\vec{\mathrm{R}}$ inside inductive ones. The syntactic restrictions will be discussed in more detail in Theorem 4.1 and their purpose will become clear from the semantics of inductive predicates presented in Section 5. To turn a $\mathfrak{I}$ expression into a predicate (in $\mathfrak{P}$) one must supply any predicate arguments; this explains the form $\mathfrak{I}(\vec{\mathfrak{P}})$. We will often omit the empty brackets $\langle\rangle$ and ( ) when inductive predicates does not use any predicate parameters. Fig. 6 gives the inductive definitions we will use to describe the linked lists and tree structures in our examples.

We will write $\text{lseg}[\vec{\mathrm{R}}, T^{\vec{\mathrm{R}}}(\cdot)]$ as shorthand for the following:

$$
\begin{array}{l}
\mu^1_{ind} \left\langle \vec{\mathrm{R}} \right\rangle\ \underline{\mathrm{I}}(x, y, \sigma). \\
\left(\begin{array}{l} \quad x = y \wedge \sigma = \emptyset \wedge \mathsf{emp} \\ \vee\ \exists nxt, \sigma' \,.\, x \mapsto T^{\vec{\mathrm{R}}}(\cdot), \_, nxt \star \underline{\mathrm{I}}(nxt, y, \sigma') \wedge nxt \neq x \wedge \sigma = \sigma' \cup \{x\} \end{array} \right)
\end{array}
$$

where $T^{\vec{\mathrm{R}}}(\cdot)$ is a formula with one expression hole and which may use

predicate variables from $\vec{\mathrm{R}}$. This allows us to conveniently describe linked lists where the data stored at each node satisfies the property $T^{\vec{\mathrm{R}}}(\cdot)$.

Recursively defined assertions: These (mutually) recursively defined assertions are the key to our work, because they let us reason naturally about challenging patterns of execution, such as self-updating code and recursion through the store. We use a similar notation as for inductively defined predicates, dropping the positive occurrence condition and the *ind* subscript:

$$\mu^k\ \mathrm{R}_1(\vec{x}_1), \ldots, \mathrm{R}_n(\vec{x}_n)\ .\ P_1, \ldots, P_n$$

The above indicates that n predicates are defined mutually recursively with arguments $\vec{x}_i$ and bodies P_i, respectively, and the superscript k indicates that the kth such predicate is selected. In order to simplify the notation for mutually recursively defined predicates we use the notation

$$(N_1, \ldots, N_n)\ :=\ \mu\ \mathrm{R}_1(x_1), \ldots, \mathrm{R}_n(x_n).\ P_1, \ldots, P_n$$

to abbreviate the following sequence of shorthand definitions:

$$\begin{aligned} N_1\ &:=\ \mu^1\ \mathrm{R}_1(x_1), \ldots, \mathrm{R}_n(x_n).\ P_1, \ldots, P_n, \\ &\vdots \\ N_n\ &:=\ \mu^n\ \mathrm{R}_1(x_1), \ldots, \mathrm{R}_n(x_n).\ P_1, \ldots, P_n \end{aligned}$$

In Section 4 we will give a grammar for formulae that can be allowed in those recursive definitions since existence of fixpoints is not automatic. We write $\mathscr{A}$ for an *allowed formula* (including in Fig. 5), i.e. one that is of an appropriate form to ensure the existence of a solution.

Remark 3.1. Note that we distinguish recursive and inductive predicates since for our semantics recursive predicates do not include inductive ones as is normally the case. This distinction is necessary here as recursive predicates can only be shown to exist for *contractive* definitions, whereas inductive definitions do not have to be contractive to exist but just monotone. This will become clearer in Section 4.

Finite sets: We use a limited first order treatment of finite sets.

4. Specification patterns for recursion through the store

We now explain the need for recursive assertions. Consider the last section of code in our main program (Fig. 2), which is responsible for disposing of the tree data structure. Let DT denote the piece of code written into the *disposeTree* cell.

Suppose we try to write a precondition for DT. This precondition must mention all the heap resources needed by DT. Firstly a tree must be present

at address t, so the precondition must include $\mathrm{tree}(t, \tau)$ for some τ. Secondly, since DT makes its recursive call through the heap at the address *disposeTree*, the precondition must include $\mathit{disposeTree} \mapsto B$ where B is a nested triple. In particular, B must state that the code stored has the same kind of behaviour as we specify for DT. But we don't have DT's specification yet, because we are still trying to formulate its precondition! It appears that we need a specification which depends on itself. Using the recursively defined predicates we can write such a specification, namely

$$\forall t. \left\{ \begin{array}{l} \exists \tau.\ \mathrm{tree}(t, \tau) \\ \star\ \mathit{DisposeTreeCode} \end{array} \right\} \cdot (t) \ \ \{\ \mathit{DisposeTreeCode}\ \}$$

where we define

$$\begin{array}{l} \mathit{DisposeTreeCode} ::= \\ \quad \mu^1\ \mathrm{R}\ .\ \mathit{disposeTree} \mapsto \forall t. \{\exists \tau. \mathrm{tree}(t, \tau) \star \mathrm{R}\} \cdot (t)\, \{\mathrm{R}\} \end{array}$$

Note that pointer *disposeTree* in the above example is assumed to be a *global constant*. Otherwise it would have to be included in the parameter list of predicate *DisposeTreeCode*. We will often use such global constants where appropriate to reduce the number of predicate arguments. This tree disposal is one particular use of recursion through the store. In the rest of this section we present a series of execution patterns which use recursion through the store, of increasing complexity. For each execution pattern we identify a corresponding specification pattern. By *pattern* we mean the shape of the specification, in particular the shape of the recursively defined assertion needed to deal with the recursion through the store.

Recursion via one or finitely many fixed pointers. The tree disposal code above shows the simplest kind of recursion through the store: a piece of code on the heap (in this case our DT code) invoking itself recursively via a pointer variable. The specification Φ_1 in Fig. 7, which generalises *DisposeTreeCode*, describes code that operates on a data structure D_1, returns a data structure D_2 and calls itself recursively through a pointer g into the heap. Note that Φ_1, like all the specifications in Figures 7 and 8, is a *predicate*, that is, belongs to syntactic category $\mathfrak{P}$. Specification Φ_2 (also in Fig. 7) describes two pieces of code on the heap that call themselves and each other recursively via two pointers g_1 and g_2. Note that the pointers g, g_1, and g_2 are fixed global constants and thus do not need to appear as arguments of Φ_1, and Φ_2, respectively. In general, Φ_n can be formulated to describe n pieces of code stored on the heap and able to call each other. (The $D, D_1, D_2, \ldots$ in Fig. 7 are metavariables, and in applications of the patterns they will be replaced by concrete formulae describing data structures.)

Note that although in this paper we will focus on proving memory safety, our patterns encompass full functional correctness specifications too. For instance, a factorial function that calls itself recursively through the store can be functionally specified using the following instance of the Φ_1 pattern:

$$\Phi_{\mathrm{Fac}}\ :=\ \mu^1 \mathrm{R}\ .\ g \mapsto \forall x, n. \{x \mapsto n \star r \mapsto \_ \star \mathrm{R}\} \cdot (x) \{x \mapsto 0 \star r \mapsto n! \star \mathrm{R}\}$$

Fixed pointers:

$$\Phi_1 := \quad \mu^1 \mathrm{R} \,.\, g \mapsto \forall \vec{x}.\{D_1 \star \mathrm{R}\} \cdot (\vec{p})\{D_2 \star \mathrm{R}\}$$

$$\Phi_2 := \quad \mu^1 \mathrm{R} \,.\, \begin{array}{l} g_1 \mapsto \forall \vec{x}_1.\{D_1 \star \mathrm{R}\} \cdot (\vec{p}_1)\{D_2 \star \mathrm{R}\} \\ \star\ g_2 \mapsto \forall \vec{x}_2.\{D_3 \star \mathrm{R}\} \cdot (\vec{p}_2)\{D_4 \star \mathrm{R}\} \end{array}$$

$$\Phi_2' := \quad \mu^1 \mathrm{R}(c,d) \,.\, \begin{array}{l} g_1 \mapsto c \ \wedge\ \forall \vec{x}_1, c', d'.\, \{D_1 \star \mathrm{R}(c',d')\}\, c(\vec{p}_1)\, \{D_2 \star \mathrm{R}(c',d')\} \\ \star\ g_2 \mapsto d \ \wedge\ \forall \vec{x}_2, c', d'.\, \{D_3 \star \mathrm{R}(c',d')\}\, d(\vec{p}_2)\, \{D_4 \star \mathrm{R}(c',d')\} \end{array}$$

With dynamic loader:

$$\Phi_{\text{withLoader}} := \quad \mathit{LoadedCode}(g_1) \star \mathit{LoadedCode}(g_2) \star \mathit{LoaderCode}$$

where

$$\begin{array}{l} (\mathit{LoadedCode}, \mathit{LoaderCode}) := \\ \quad \mu\, \mathrm{R}(a), \mathrm{S} \,. \\ \quad\quad a \mapsto \forall \vec{x}.\, \{D \star \mathrm{R}(g_1) \star \mathrm{R}(g_2) \star \mathrm{S}\} \cdot (\vec{p})\, \{D \star \mathrm{R}(g_1) \star \mathrm{R}(g_2) \star \mathrm{S}\}\,, \\ \quad\quad \mathit{loader} \mapsto \forall a, \mathit{ID}.\, \{a \mapsto \_\} \cdot (a, \mathit{ID})\, \{\mathrm{R}(a)\} \end{array}$$

List of code:

$$\mathit{CLseg}_1 := \quad \mu^1 \mathrm{R}(x,y,\sigma) \,.\, \mathrm{lseg}[\mathrm{S}, T_1^S(\cdot)](\mathrm{R})(x,y,\sigma)$$

where $T_1^S(\cdot)$ is:

$$\forall \vec{x}.\, \{\exists a, \sigma.\, D \star \mathit{header} \mapsto a \star \mathrm{S}(a, \mathsf{null}, \sigma)\} \cdot (\vec{p})\, \{\exists a, \sigma.\, D \star \mathit{header} \mapsto a \star \mathrm{S}(a, \mathsf{null}, \sigma)\}$$

List of code (with updates restricted):

$$\mathit{CLseg}_2 := \quad \mu^1 \mathrm{R}(x,y,\sigma) \,.\, \mathrm{lseg}[\mathrm{S}, T_2^S(\cdot)](\mathrm{R})(x,y,\sigma)$$

where $T_2^S(\cdot)$ is:

$$\forall \vec{x}, a, \sigma.\, \{D \star \mathit{header} \mapsto a \star \mathrm{S}(a, \mathsf{null}, \sigma)\} \cdot (\vec{p})\, \{D \star \mathit{header} \mapsto a \star \mathrm{S}(a, \mathsf{null}, \sigma)\}$$

Figure 7: Specification patterns for recursion through the store.

Similar specifications were used in [10] to reason about the mutual recursion arising between a recursive function implementation and a generic memoiser for recursive functions. Also similar specifications were used in [18] to reason about runtime updates to a server.

Note that specification Φ_1 allows the code pointed to by g to update itself (like our OSCILLATE example), as long as the update is with code which behaves in the same way. This is because the precondition and postcondition simply state that code with the required behaviour is present; they do not insist it be the same code in the poststate as in the prestate. Similarly, in Φ_2 the two pieces of code can update themselves *and each other*, as long as they do

Data structure with pointers:

$$CLseg_3 := \quad \mu^1 \mathrm{R}(x,y,\sigma)\ .\ \mathrm{lseg}[\mathrm{S}, T_3^S(\cdot)](\mathrm{R})(x,y,\sigma)$$

where $T_3^S(\cdot)$ is

$$\forall \vec{x}\ .\ \left\{ \begin{array}{l} \exists a, \sigma, \tau.\ \tau \subseteq \sigma\ \wedge\ D(\tau) \\ \star\ header \mapsto a\ \star\ \mathrm{S}(a, \mathsf{null}, \sigma) \end{array} \right\}\ \cdot (\vec{p})\ \left\{ \begin{array}{l} \exists a, \sigma, \tau.\ \tau \subseteq \sigma\ \wedge\ D(\tau) \\ \star\ header \mapsto a\ \star\ \mathrm{S}(a, \mathsf{null}, \sigma) \end{array} \right\}$$

Data structure with pointers and a dynamic loader:

$$\begin{array}{l} (CLseg_4, LoaderCode) ::= \\ \quad \mu\ \mathrm{R}_1(x,y,\sigma), \mathrm{R}_2\ . \\ \qquad \mathrm{lseg}[\mathrm{S}_1, \mathrm{S}_2, T_4^{S_1,S_2}(\cdot)](\mathrm{R}_1, \mathrm{R}_2)(x,y,\sigma), \\ \qquad loader \mapsto \forall codeID, code\_addr\_addr, \sigma_1\ . \\ \qquad\quad \left\{ \begin{array}{l} \exists a\ . \\ \quad header \mapsto a \\ \star\ \mathrm{R}_1(a, \mathsf{null}, \sigma_1) \\ \star\ code\_addr\_addr \mapsto \_ \end{array} \right\}\ \cdot (codeID, \\ \qquad\quad code\_addr\_addr)\ \left\{ \begin{array}{l} \exists a, r, \sigma_2\ .\ \sigma_1 \cup \{r\} \subseteq \sigma_2 \\ \quad \wedge\ header \mapsto a \\ \quad \star\ \mathrm{R}_1(a, \mathsf{null}, \sigma_2) \\ \quad \star\ code\_addr\_addr \mapsto r \end{array} \right\} \end{array}$$

where $T_4^{S_1,S_2}(\cdot)$ is

$$\forall \vec{x}\ .\ \left\{ \begin{array}{l} \exists a, \sigma, \tau\ .\ \tau \subseteq \sigma\ \wedge \\ \quad D(\tau) \star header \mapsto a \\ \star\ \mathrm{S}_1(a, \mathsf{null}, \sigma) \\ \star\ \mathrm{S}_2 \end{array} \right\}\ \cdot (\vec{p})\ \left\{ \begin{array}{l} \exists a, \sigma, \tau\ .\ \tau \subseteq \sigma\ \wedge \\ \quad D(\tau) \star header \mapsto a \\ \star\ \mathrm{S}_1(a, \mathsf{null}, \sigma) \\ \star\ \mathrm{S}_2 \end{array} \right\}$$

Figure 8: More specification patterns for recursion through the store.

so appropriately. This is good as it accommodates clever uses of higher-order store, but there are also occasions when it is necessary to explicitly disallow update. This happens when one has a public and a (stronger) private specification for some code; allowing "external" updates to the code might preserve only the public specification. See Section 9.5 in [19] for an example. In this case the specification needs to state explicitly that the content of the code remains the same; this is what Φ_2' does. Here, again, pointer variables g_1 and g_2 are fixed (global variables) but Φ_2' needs two arguments for the code stored in those pointers (c and d, respectively) which could change in principle. In the following patterns all changeable values will be represented as arguments to the specifications (predicates) in question. Where we assume constant values we use global variables.

Usage with a dynamic loader. As we pointed out, the preceding specifications permit in place update of code. This treats behaviour like that of our OSCILLATE module, which explicitly writes code onto the heap; but it does not account for behaviour like that of LOAD_OVERWRITE, where a loader function of the main program is invoked to load other code that is required.

The specification $\Phi_{\text{withLoader}}$ in Fig. 7 describes a situation where two pieces of code are on the heap, calling themselves and each other recursively; but each may also call a loader procedure provided by the main program. Note the asymmetry in the specification of the loader, which could not be expressed using the invariant extension operator $\otimes$: R appears in the postcondition but nowhere in the precondition.

Recursion via a list of code. The next step up in complexity is where a linked list is used to hold an arbitrary number of pieces of code. We suppose that each list node has three fields: the code, an ID number identifying the code, and a next pointer. The ID numbers allow the pieces of code to locate each other by searching through the list. We suppose that the cell pointed to by global constant *header* contains a pointer to the start of the list. To reason about this setup, we use the $\text{lseg}[\vec{\text{R}}, T^{\vec{\text{R}}}(\cdot)]$ shorthand, from page 13, to define recursively a predicate $CLseg_1$ (Fig. 7) for segments of code lists. We point out the similarity between these idealised code lists and for example the *net_device* list that the Linux kernel uses to manage dynamically loaded and unloaded network device drivers [20].

Note that the pieces of code are free to extend or update the code list in any way they like, e.g. by updating themselves or adding or updating other code, as long as any new code also behaves again in the same way. The existential quantifiers over a and σ in the auxiliary $T_1(\cdot)$ are needed to allow for updates that might change the layout of the list in memory.

A variation restricting code updates. One can vary the above specification in several ways. For instance, we can allow the pieces of code in the list to call each other and update each other in-place but prohibit them from changing the shape of the list in memory. The predicate $CLseg_2$ (Fig. 7) does this by a simple change of quantifiers, requiring that the starting point a and locations σ of the list are the same before and after each piece of code runs.

Recursion via a set of pointers stored in a data structure. In the setup described above with $CLseg_1$ and $CLseg_2$, the pieces of code in the list found each other using the ID numbers in the list structure. But instead, the program might set up code pointers referencing code in such a list so that the pieces of code can invoke each other directly. We suppose that these direct code pointers live in the data structure D, writing $D(\tau)$ for a data structure whose code pointers collectively point to the set of addresses τ. The recursive specification we need is $CLseg_3$ given in Fig. 8; the constraint $\tau \subseteq \sigma$ says that all code pointers in D must point into the code list.

Code lists with pointer structures and a dynamic loader. The most complex pattern we will look at, $CLseg_4$ in Fig. 8, combines the above idea with the use of a dynamic loader. This gives the kind of execution pattern found in our example program, where the data structure $D(\cdot)$ will be $\text{tree}(\mathit{tree}, \cdot)$.

4.1. The master pattern

The specification patterns described above all use the fixpoint operator μ to build recursively defined predicates. But one must wonder whether this is

meaningful, since arbitrary functions need not have fixpoints. To ensure the well-definedness of our specifications, we must show that the required fixpoints actually exist.

Performing such existence arguments on an *ad hoc* basis is undesirable, particularly because some of the specifications feature recursive definitions using μ intertwined with inductive definitions using μ_{ind}. To address this, we now present a "*master pattern*" describing a whole class of specifications which can be shown to exist. The appropriate existence proof will be given in Section 5. The master pattern covers all the specifications in Figures 7 and 8, and all others we have encountered a need for, with the caveat that we are not considering higher-order logic specifications.

Theorem 4.1. The master pattern. (Mutually) recursively defined predicates are guaranteed to exist when they take the form

$$\mu^k \, \mathrm{R}_1(\vec{x}_1), \ldots, \mathrm{R}_n(\vec{x}_n) \,.\, \mathscr{A}_1, \ldots, \mathscr{A}_n$$

where $\mathscr{A}_i$ comes from the following grammar:

$$\begin{array}{rll}
\mathscr{A} ::= & \mathscr{A} \wedge \mathscr{A} \mid \mathscr{A} \vee \mathscr{A} \mid \mathscr{A} \star \mathscr{A} & \\
& \mid \forall x.\mathscr{A} \mid \exists x.\mathscr{A} & \\
& \mid e \mapsto e \mid \mathsf{emp} \mid e = e \mid e < e & \\
& \mid \{\Phi\}\, e(\vec{e})\, \{\Phi\} & (\Phi\text{s are arbitrary formulae}) \\
& \mid I(\vec{\mathrm{R}})(\vec{e}) & (I \in \mathfrak{I} \text{ is contractive in its predicate arguments})
\end{array}$$

A sufficient syntactic condition for

$$\mu^k_{ind} \left\langle \vec{\mathrm{R}} \right\rangle \, \underline{\mathrm{I}}_1(\vec{x}_1), \ldots, \underline{\mathrm{I}}_n(\vec{x}_n).\ P_1, \ldots, P_n$$

to be contractive in $\vec{\mathrm{R}}$ in addition to ensuring that the definition gives rise to an inductive predicate is that each P_i comes from the following grammar:

$$\begin{array}{rll}
\mathscr{B} ::= & \mathscr{B} \wedge \mathscr{B} \mid \mathscr{B} \vee \mathscr{B} \mid \mathscr{B} \star \mathscr{B} & \\
& \mid \exists x.\mathscr{B} & \\
& \mid e \mapsto e \mid \mathsf{emp} \mid e = e \mid e < e & \\
& \mid \forall \vec{x}.\ \{\Phi_1\}\, e(\vec{e})\, \{\Phi_2\} & (\vec{\underline{\mathrm{I}}} \notin \Phi_i, i = 1, 2) \\
& \mid \underline{\mathrm{I}}(\vec{e}) & (\underline{\mathrm{I}} \text{ is one of } \underline{\mathrm{I}}_1, \ldots, \underline{\mathrm{I}}_N)
\end{array}$$

□

Note that the parameter predicates R can occur in an $\mathscr{B}$ formula only within the assertions Φ_1 and Φ_2 of a triple definition. The reason for the particular shape of these grammars is to establish syntactically an approximation to semantic contractiveness in the sense of (ultra)metric spaces. The denotations of assertions are predicates on heaps (indexed by worlds that represent invariants added on by $\otimes$) which can be endowed with a distance function such that they can be regarded as non-empty one-bounded ultrametric spaces (see Sect. 5). Any contractive function on such spaces (or here predicates) must have a fixpoint

by Banach's fixpoint theorem (see also e.g. [21]). Since our recursive definitions also involve inductive ones, we have made the occurrences of recursively defined predicates in inductive ones visible via the $\langle\ \rangle$ notation. So we actually define families of inductive predicates which exist due to the assumption that the inductive definitions only use positive occurrences of the inductive predicates. The grammar for $\mathscr{B}$ ensures now that the predicates defined are inductive *and* that they are contractive in their parameters. Details will be discussed in Section 5.

Remark 4.2. Unlike the "*formal contractiveness*" of [12] that includes the $\otimes$ operator our master pattern does not include $\otimes$. The reason is that we do not have a distribution rule for $\otimes$ (like those in Fig. 10) over recursive definitions, i.e. the following is *not* an axiom

$$(\mu^k P_1 \cdots P_n.\, \Phi_1 \cdots \Phi_n) \otimes R \iff \mu^k P_1 \cdots P_n.\, \Phi_1 \cdots (\Phi_k \otimes R) \cdots \Phi_n$$

Yet, we conjecture that its effect on recursive predicates can be expressed by unfolding the recursive definition and using the distributions axioms for $\otimes$ as found in [9, 12]. The details will appear elsewhere. In case, however, the left hand side P of $P \otimes R$ is not recursively defined itself we can already express the effect of the tensor $\otimes$ for any concrete instance. For example, one can still express the specification

$$\mu^1\ \mathrm{R}.\ x \mapsto \{y \mapsto \{a \mapsto \_\} \cdot \{a \mapsto \_\}\} \cdot \{y \mapsto \{a \mapsto \_\} \cdot \{a \mapsto \_\}\} \otimes \mathrm{R}$$

by writing

$$\mu^1\ \mathrm{R}, \mathrm{Y}.\ x \mapsto \{\mathrm{R} \star \mathrm{Y}\} \cdot \{\mathrm{R} \star \mathrm{Y}\},\quad y \mapsto \{a \mapsto \_ \star \mathrm{R}\} \cdot \{a \mapsto \_ \star \mathrm{R}\}$$

and the proof of this equivalence uses the recursion rules and the distribution rules for $\otimes$ as seen in Figures 11 and 10, respectively. □

5. Proof rules and Soundness

In this section we state the proof rules we use with our assertion language, and prove their soundness. We also show that our recursively defined predicates exist, ie. we give a proof of Theorem 4.1.

5.1. Proof Rules

Our formal proof rules make use of the formal judgment $\Gamma \vdash \{P\}\,`C'\,\{Q\}$ stating that in variable context $\Gamma$ the *assertion* $\{P\}\,`C'\,\{Q\}$ can be derived for a command expression C. Since universal validity of assertion $\{P\}\,`C'\,\{Q\}$ coincides with the common interpretation of $\{P\}\,C\,\{Q\}$ (see [9, 12]) we can identify them and thus often drop the quotes around commands in top level triples.

Since we use an untyped language, the variable context Γ simply consists of a list of distinct variables that can appear on the right hand side of $\vdash$.

$$\textsc{Deref}\quad \frac{\Gamma, x \vdash \{P \star e \mapsto x\}\,\text{`}C\text{'}\,\{Q\}}{\Gamma \vdash \{\exists x.(P \star e \mapsto x)\}\,\text{`}\mathsf{let}\ x = [e]\ \mathsf{in}\ C\text{'}\,\{Q\}}(x \notin fv(e, Q))$$

$$\textsc{Update}\quad \frac{}{\Gamma \vdash \{e \mapsto \_ \star P\}\,\text{`}[e] := e_0\text{'}\,\{e \mapsto e_0 \star P\}}$$

$$\textsc{New}\quad \frac{\Gamma, x \vdash \{P \star x \mapsto \vec{e}\}\,\text{`}C\text{'}\,\{Q\}}{\Gamma \vdash \{P\}\,\text{`}\mathsf{let}\ x = \mathsf{new}\ \vec{e}\ \mathsf{in}\ C\text{'}\,\{Q\}}(x \notin fv(P, \vec{e}, Q))$$

$$\textsc{Free}\quad \frac{}{\Gamma \vdash \{e \mapsto \_ \star P\}\,\text{`}\mathsf{free}\ e\text{'}\,\{P\}} \qquad \textsc{Skip}\quad \frac{}{\Gamma \vdash \{P\}\,\text{`}\mathsf{skip}\text{'}\,\{P\}}$$

$$\textsc{Seq}\quad \frac{\Gamma \vdash \{P\}\,\text{`}C\text{'}\,\{R\} \qquad \Gamma \vdash \{R\}\,\text{`}D\text{'}\,\{Q\}}{\Gamma \vdash \{P\}\,\text{`}C; D\text{'}\,\{Q\}}$$

$$\textsc{If}\quad \frac{\Gamma \vdash \{P \wedge e_0 = e_1\}\,\text{`}C\text{'}\,\{Q\} \qquad \Gamma \vdash \{P \wedge e_0 \neq e_1\}\,\text{`}D\text{'}\,\{Q\}}{\Gamma \vdash \{P\}\,\text{`}\mathsf{if}\ (e_0 = e_1)\ \mathsf{then}\ C\ \mathsf{else}\ D\text{'}\,\{Q\}}$$

Figure 9: Proof rules for program statements.

Most of the rules we use are exactly the same as in [9] where they have been proved sound. They can be found in Fig. 9 (rules for program statements other than eval) and Fig. 10 (distribution laws for $\otimes$, other than for $\{P\}\,e(\vec{e})\,\{Q\} \otimes R$). The new proof rules, including rules for running stored code and for the λ binder, are stated in Fig. 11. There are two groups of new rules: a large number of rules (for instance (Lambda), (Conseq) and (ForallExists)) are just adaptations of the rules of [9, 12] to the fact that our procedures now have arguments. The adaptation of the distribution rule for triples $\otimes$-TripleDist to procedures with arguments uses the $\_ \circ \_$ notation where $R \circ R'$ abbreviates $(R \otimes R') \star R'$.

A second group of rules in Fig. 11 deals with the extended recursive definitions. Rule (Mu) allows for folding and unfolding mutual recursive definitions and there is a rule for inductive definitions, (MuInd), as well, as described earlier. For inductive definitions we need an induction rule (Induction)[2]. The induction rule can be used to prove statements $\Phi[\mathrm{X} \backslash \mu_{\mathrm{ind}}\ \underline{\mathrm{I}}(\vec{x}).\ \ldots]$ provided Φ

[2]We do not give a rule for mutual induction here as we won't need it for our examples, but this could be given analogously.

$$(\kappa x.P) \otimes R \Leftrightarrow \kappa x.(P \otimes R) \quad (\kappa \in \{\forall, \exists\}, x \notin fv(R))$$

$$(P \otimes R) \otimes R' \Leftrightarrow P \otimes (R \circ R')$$

$$(P \oplus Q) \otimes R \Leftrightarrow (P \otimes R) \oplus (Q \otimes R) \quad (\oplus \in \{\Rightarrow, \wedge, \vee, \star\})$$

$$P \otimes R \Leftrightarrow P \quad (\text{P is one of True, False, } \mathsf{emp},\ e = e' \text{ or } e \mapsto \vec{e})$$

Figure 10: Distribution axioms for $\otimes$ (other than for $\{P\}\, e(\vec{e})\, \{Q\} \otimes R$).

is an "*elimination formula*" of the form

$$\forall \vec{x}.\, \mathrm{X}(\vec{e}) \star \Phi_0 \Rightarrow \Phi_1$$

for an inductively defined predicate X. Formulae Φ_0 and Φ_1 must not contain variable X but can contain $\vec{x}$ and $\underline{\mathrm{I}}$. The restriction of the formulae Φ is necessary to guarantee later that they are admissible in X. Any admissible Φ would be suitable but for the proofs in this paper the above "*elimination formulae*" are sufficient.

5.2. Soundness of Proof Rules

We now address the soundness of our rules. Instead of giving semantics directly to our programming language, we give a translation $\mathrm{tr}(-)$ into the programming language used in [9] ; this translation induces the semantics of our programs. Assertions are similarly translated into the logic of [9] extended by finite sets and explicit recursion as described in Fig. 5. The soundness of those extensions is discussed at the end of this section. The translation $\mathrm{tr}(-)$ is as follows.

Definition 5.1. Translation of programs. Fix a large number N and distinguished variables $\vec{w} = w_1, \ldots, w_N$ never used in our (untranslated) programs. The translation of expressions and commands is *structural* except for two cases:

$$\mathrm{tr}(\text{`}\lambda x_1, \ldots, x_n.C\text{'}) \quad := \quad \text{`}\mathsf{let}\ x_1 = [w_1]\ \mathsf{in}\ \ldots \mathsf{let}\ x_n = [w_n]\ \mathsf{in}\ \mathrm{tr}(C)\text{'}$$

$$\mathrm{tr}(\mathsf{eval}\ [e](e_1, \ldots, e_n)) \quad := \quad [w_1] := \mathrm{tr}(e_1)\ ;\ \ldots\ ;\ [w_n] := \mathrm{tr}(e_n)\ ;\ \mathsf{eval}\ [\mathrm{tr}(e)]$$

By "structural" we mean that the translation simply propagates itself towards the syntactic leaves, being simply applied to *all* constituents, as in $\mathrm{tr}(e_1 + e_2) = \mathrm{tr}(e_1) + \mathrm{tr}(e_2)$ and

$$\mathrm{tr}(\mathsf{let}\ x = [e]\ \mathsf{in}\ C) \quad = \quad \mathsf{let}\ \mathrm{tr}(x) = [\mathrm{tr}(e)]\ \mathsf{in}\ \mathrm{tr}(C) \qquad \square$$

Definition 5.2. Translation of assertions and proof obligations. The translation of assertions is structural except for the clause for Hoare triples, where

$$\mathrm{tr}(\{P\}\, e(\vec{e})\, \{Q\}) \quad := \quad \{\mathrm{tr}(P) \star W(\mathrm{tr}(\vec{e}))\}\, \mathrm{tr}(e)\, \{\mathrm{tr}(Q) \star W\} \qquad \text{with}$$

EVAL
$$\frac{\Gamma, k \vdash R(k) \Rightarrow \{P \star e \mapsto R(\cdot)\}\, k(e_1, \ldots, e_n)\, \{Q\}}{\Gamma \vdash \{P \star e \mapsto R(\cdot)\}\, \text{`eval } [e](e_1, \ldots, e_n)\text{'}\, \{Q\}}$$

LAMBDA
$$\frac{\Gamma, \vec{x} \vdash \{P[\vec{a}\backslash\vec{x}]\}\, \text{`}C\text{'}\, \{Q[\vec{a}\backslash\vec{x}]\}}{\Gamma \vdash \{P\}\, \text{`}\lambda\vec{x}.C\text{'}(\vec{a})\, \{Q\}}$$
(variables in $\vec{x}, \vec{a}$ all distinct, and $\vec{a}$ disjoint from $fv(C)$)

FORALLEXISTS
$$\Gamma \vdash \forall x.\, \{P\}\, e(\vec{e})\, \{Q\} \Rightarrow \{\exists x.P\}\, e(\vec{e})\, \{\exists x.Q\} \quad (x \notin fv(e, \vec{e}))$$

CONSEQ
$$\frac{\Gamma \vdash P' \Rightarrow P \qquad \Gamma \vdash Q \Rightarrow Q'}{\Gamma \vdash \{P\}\, e(\vec{e})\, \{Q\} \Rightarrow \{P'\}\, e(\vec{e})\, \{Q'\}}$$

$\star$-FRAME
$$\Gamma \vdash \{P\}\, e(\vec{e})\, \{Q\} \Rightarrow \{P \star R\}\, e(\vec{e})\, \{Q \star R\}$$

$\otimes$-FRAME
$$\frac{\Gamma \vdash P}{\Gamma \vdash P \otimes R}$$

$\otimes$-TRIPLEDIST
$$\Gamma \vdash \{P\}\, e(\vec{e})\, \{Q\} \otimes R \Leftrightarrow \{P \circ R\}\, e(\vec{e})\, \{Q \circ R\}$$

MU
$$(\mu^k\ \mathrm{R}_1(\vec{x}_1), \ldots, \mathrm{R}_n(\vec{x}_n)\ .\ \mathscr{A}_1, \ldots, \mathscr{A}_n)(\vec{e})$$

$$\Leftrightarrow$$

$$\Gamma \vdash \mathscr{A}_k \left[\begin{array}{lcl} \mathrm{R}_1 & & \mu^1\ \mathrm{R}_1(\vec{x}_1), \ldots, \mathrm{R}_n(\vec{x}_n)\ .\ \mathscr{A}_1, \ldots, \mathscr{A}_n, \\ \ldots, & \backslash & \ldots, \\ \mathrm{R}_n & & \mu^n\ \mathrm{R}_1(\vec{x}_1), \ldots, \mathrm{R}_n(\vec{x}_n)\ .\ \mathscr{A}_1, \ldots, \mathscr{A}_n \end{array}\right] [\vec{x}_k\backslash\vec{e}]$$

MUIND
$$(\mu^k_{ind}\ \langle \mathrm{R}_1, \ldots, \mathrm{R}_N\rangle\ \underline{\mathrm{I}}_1(\vec{x}_1), \ldots, \underline{\mathrm{I}}_n(\vec{x}_n)\ .\ P_1, \ldots, P_n)[\mathrm{S}_1, \ldots, \mathrm{S}_N](\vec{e})$$

$$\Leftrightarrow$$

$$\Gamma \vdash P_k \left[\begin{array}{lcl} \underline{\mathrm{I}}_1 & & (\mu^1_{ind}\ \left\langle \vec{\mathrm{R}_i} \right\rangle\ \underline{\mathrm{I}}_1(\vec{x}_1), \ldots, \underline{\mathrm{I}}_n(\vec{x}_n)\ .\ P_1, \ldots, P_n)(\vec{\mathrm{S}_i}), \\ \ldots, & & \ldots, \\ \underline{\mathrm{I}}_n & & (\mu^n_{ind}\ \left\langle \vec{\mathrm{R}_i} \right\rangle\ \underline{\mathrm{I}}_1(\vec{x}_1), \ldots, \underline{\mathrm{I}}_n(\vec{x}_n)\ .\ P_1, \ldots, P_n)(\vec{\mathrm{S}_i}), \\ \mathrm{R}_1, & \backslash & \mathrm{S}_1, \\ \ldots, & & \ldots, \\ \mathrm{R}_N & & \mathrm{S}_N \end{array}\right] [\vec{x}_k\backslash\vec{e}]$$

INDUCTION
$$\frac{\Gamma \vdash \Phi[\mathrm{X}\backslash\lambda\vec{x}.\,\mathrm{False}] \qquad \Gamma, \mathrm{X} \vdash \Phi \ \Rightarrow\ \Phi[\mathrm{X}\backslash P[\underline{\mathrm{I}}, \vec{\mathrm{R}}\backslash\mathrm{X}, \vec{\mathrm{S}}]]}{\Gamma \vdash \Phi[X\backslash\mu^1_{ind} <\vec{\mathrm{R}}> \underline{\mathrm{I}}(\vec{x}).\, P)(\vec{S})]}$$
$\vec{R} \notin fv(\vec{S}, \Phi),\ X \in fv(\Phi)$
Φ elimination formula

Figure 11: Proof rules for our logic which differ from those of [9].

$$W(e_1,\ldots,e_n) \quad := \quad w_1 \mapsto e_1 \star \cdots \star w_n \mapsto e_n \star w_{n+1} \mapsto \_ \star \cdots \star w_N \mapsto \_$$

(writing just W when n=0). For proof obligations we define $\mathrm{tr}(\Gamma \vdash P) := \Gamma, \vec{w} \vdash \mathrm{tr}(P)$. □

We have proved the soundness of all the rules in Fig. 11: for each rule with premise P and conclusion Q, we proved $\mathrm{tr}(Q)$ from $\mathrm{tr}(P)$ in the logic of [9]. Two such proofs follow.

Theorem 5.3. Rule (EVAL) in Fig. 11 holds.

Proof. The conclusion of the rule translates to

$$\Gamma, \vec{w} \vdash \ \{\mathrm{tr}(P) \star \mathrm{tr}(e) \mapsto \mathrm{tr}(R(\cdot)) \star W\}\, \mathrm{tr}(\text{`}\mathsf{eval}\ [e](e_1,\ldots,e_n)\text{'})\, \{\mathrm{tr}(Q) \star W\}$$

which expands to

$$\Gamma, \vec{w} \vdash \ \{\mathrm{tr}(P) \star \mathrm{tr}(e) \mapsto \mathrm{tr}(R(\cdot)) \star W\}\, \text{`}C\text{'}\, \{\mathrm{tr}(Q) \star W\}$$

where C is $[w_1] := \mathrm{tr}(e_1)\ ; \ldots ;\ [w_n] := \mathrm{tr}(e_n)\ ;\ \mathsf{eval}\ [\mathrm{tr}(e)]$. Using the rules for sequential composition and heap write, this will follow from

$$\Gamma, \vec{w} \vdash \\ \{\mathrm{tr}(P) \star \mathrm{tr}(e) \mapsto \mathrm{tr}(R(\cdot)) \star W(\mathrm{tr}(e_1),\ldots,\mathrm{tr}(e_n))\}\, \mathsf{eval}\ [\mathrm{tr}(e)]\, \{\mathrm{tr}(Q) \star W\}$$

By the (EVAL) rule of [9], this follows from $\Gamma, \vec{w}, k \vdash$

$$\mathrm{tr}(R(k)) \Rightarrow \{\mathrm{tr}(P) \star W(\mathrm{tr}(e_1),\ldots,\mathrm{tr}(e_n)) \star \mathrm{tr}(e) \mapsto \mathrm{tr}(R(\cdot))\}\, k\, \{\mathrm{tr}(Q) \star W\}$$

but this is the same as what one gets when translating the premise of our prospective rule, so we are done. □

Theorem 5.4. ($\otimes$-TRIPLEDIST) in Fig. 11 holds.

Proof. Expanding the $\circ$ abbreviation and translating the axiom gives

$$\Gamma, \vec{w} \vdash \quad \Leftrightarrow \quad \begin{array}{l} \{\mathrm{tr}(P) \star W(\mathrm{tr}(\vec{e}))\}\, \mathrm{tr}(e)\, \{\mathrm{tr}(Q) \star W\} \otimes \mathrm{tr}(R) \\ \{\mathrm{tr}((P \otimes R) \star R) \star W(\mathrm{tr}(\vec{e}))\}\, \mathrm{tr}(e)\, \{\mathrm{tr}((Q \otimes R) \star R) \star W\} \end{array} \tag{1}$$

We rewrite the right hand side of this using, in successive steps, 1. the monoid properties of $\star$, 2. the fact that for all A and all $\vec{e}$, $W(\vec{e}) \otimes A \Leftrightarrow W(\vec{e})$, and 3. the distribution axiom $(A \star B) \otimes C \Leftrightarrow (A \otimes C) \star (B \otimes C)$:

$$\begin{array}{ll} & \{(\mathrm{tr}(P) \otimes \mathrm{tr}(R)) \star \mathrm{tr}(R) \star W(\mathrm{tr}(\vec{e}))\}\, \mathrm{tr}(e)\, \{(\mathrm{tr}(Q) \otimes \mathrm{tr}(R)) \star \mathrm{tr}(R) \star W\} \\ \Leftrightarrow & \{(\mathrm{tr}(P) \otimes \mathrm{tr}(R)) \star W(\mathrm{tr}(\vec{e})) \star \mathrm{tr}(R)\}\, \mathrm{tr}(e)\, \{(\mathrm{tr}(Q) \otimes \mathrm{tr}(R)) \star W \star \mathrm{tr}(R)\} \\ \Leftrightarrow & \{(\mathrm{tr}(P) \otimes \mathrm{tr}(R)) \star (W(\mathrm{tr}(\vec{e})) \otimes \mathrm{tr}(R)) \star \mathrm{tr}(R)\} \\ & \quad \mathrm{tr}(e) \\ & \{(\mathrm{tr}(Q) \otimes \mathrm{tr}(R)) \star (W \otimes \mathrm{tr}(R)) \star \mathrm{tr}(R)\} \\ \Leftrightarrow & \{(\mathrm{tr}(P) \star W(\mathrm{tr}(\vec{e}))) \otimes \mathrm{tr}(R)) \star \mathrm{tr}(R)\}\, \mathrm{tr}(e)\, \{(\mathrm{tr}(Q) \star W) \otimes \mathrm{tr}(R)) \star \mathrm{tr}(R)\} \\ \Leftrightarrow & \{(\mathrm{tr}(P) \star W(\mathrm{tr}(\vec{e}))) \circ \mathrm{tr}(R)\}\, \mathrm{tr}(e)\, \{(\mathrm{tr}(Q) \star W) \circ \mathrm{tr}(R)\} \end{array}$$

Thus (1) is equivalent to

$$\begin{array}{lll} \Gamma, \vec{w} \vdash & & \{\mathrm{tr}(P) \star W(\mathrm{tr}(\vec{e}))\}\, \mathrm{tr}(e)\, \{\mathrm{tr}(Q) \star W\} \otimes \mathrm{tr}(R) \\ & \Leftrightarrow & \{(\mathrm{tr}(P) \star W(\mathrm{tr}(\vec{e}))) \circ \mathrm{tr}(R)\}\, \mathrm{tr}(e)\, \{(\mathrm{tr}(Q) \star W) \circ \mathrm{tr}(R)\} \end{array}$$

and this is an instance of the corresponding axiom for the language of [9], which is

$$\Gamma \vdash \quad \{P\}\, e\, \{Q\} \otimes R \quad \Leftrightarrow \quad \{P \circ R\}\, e\, \{Q \circ R\}$$

□

The above proofs consisted of a few reasoning steps to reduce the (translated) rule for our language into the corresponding rule from [9]. This scheme works for all the proof rules except (LAMBDA) – which of course has no equivalent in [9] – whose proof is not difficult. In fact, only the proof rules that involve triples, eval or λ need to be checked at all, because the translation is structural for all other logical constructs. For example, the translation of the propositional law $P \Rightarrow (P \vee Q)$ is simply $\mathrm{tr}(P) \Rightarrow (\mathrm{tr}(P) \vee \mathrm{tr}(Q))$, which is a substitution instance of the original law.

We must also show that the required extensions to the logic of [9], and their associated proof rules, can be soundly constructed. We now describe how this is done.

5.3. Soundness of extensions

Inductive predicates and finite sets. In order to specify lists and trees we use sets of values that e.g. describe which values are stored in a tree or a list. Set expressions are represented by Σ in the BNF grammar of Figure 5. In fact, those sets will always be finite. They are not used by the programming language nor stored on the heap and can therefore receive a natural finite set semantics. The crucial subset proposition on sets can be interpreted canonically as follows where σ and τ are sets (of integers[3]):

$$[\![\sigma \subseteq \tau]\!]_\eta \stackrel{def}{=} \{\ h \in \mathit{Heap} \mid [\![\sigma]\!]_\eta \subseteq [\![\tau]\!]_\eta\ \}$$

Note that the meaning of this predicate does not depend on any current invariant (world in the terminology of [9]) and is "pure" in the sense that it either is *Heap* (the meaning of True) or $\emptyset$ (the meaning of False). For that reason, the interpretation of subset assertions is canonically admissible and uniform as required by the semantics in [9, 12] where one can also find the full definitions of the semantics we are extending. In this section we intend to abstract away from as many of the details as possible.

We need to show that the semantics of [9] can be extended by inductive and recursive predicates (and combinations thereof). In the following *UAdm*

[3]This is sufficient for our purposes as pointers are integers. For more complex data types the sets would have to be extended to be able to contain more general data.

will denote the uniform and admissible subsets of *Heap* the domain of higher-order stores. Moreover *Pred* denotes the non-expansive maps from a convenient domain of (recursively defined) worlds W that represent the implicit invariants that have been collected by invariant extension (for details see [12]) to *UAdm*. Due to its extra properties, *UAdm*, and therefore also *Pred*, can be endowed with a distance function such that *Pred* can be shown to be non-empty complete ultrametric (1-bounded) spaces. This, in turn, means that contractive maps $Pred \to Pred$ are guaranteed to have a fixpoint by Banach's fixpoint theorem (see [21]).

5.4. Existence of inductive assertions

The semantics of inductively defined predicates $\vec{\mathrm{I}}$, parameterised by recursive predicates $\vec{\mathrm{R}}$,

$$\mu^k_{ind} \left\langle \vec{\mathrm{R}} \right\rangle \ \underline{\mathrm{I}}_1(\vec{x_1}), \ldots, \underline{\mathrm{I}}_n(\vec{x_n}).\ P_1, \ldots, P_n$$

needs to be established.

To that end, we first define $\mathrm{arg} \stackrel{def}{=} \sum_{i \in \{1,\ldots n\}} \mathsf{ValS}^{\kappa_i}$ where ValS is the union of Val and finite sets and κ_i is the arity of $\underline{\mathrm{I}}_i$, ie. the length of $\vec{x}_i$. Predicate definitions depend on a predicate environment Ξ that is supposed to contain definitions for the arguments $\vec{\mathrm{R}}$[4], i.e. $\mathrm{dom}(\Xi) = \vec{\mathrm{R}}$. Since we often use global constants in definitions, predicate definitions also depend on an environment of first-class values η. We can then define a functional $\Psi_{\eta,\Xi} : Pred^{\mathrm{arg}} \to Pred^{\mathrm{arg}}$ in a pointwise fashion as follows:

$$\Psi_{\eta,\Xi}(F)(k, \vec{v}) \stackrel{def}{=} [\![P_k]\!]_{\eta[\vec{x_k} \mapsto \vec{v}],\ \Xi[\underline{\mathrm{I}}_i \mapsto \lambda \vec{y}_i .\, F(i, \vec{y}_i)]} \tag{2}$$

where $|\vec{v}|$ is the arity κ_k of $\underline{\mathrm{I}}_k(\vec{x}_k)$ and $|\vec{y}_i|$ is the arity κ_i of $\underline{\mathrm{I}}_i(\vec{x}_i)$.

In the following, Ω denotes the minimal element of $Pred^{\mathrm{arg}}$, ie. $\lambda \vec{x}.\, [\![\mathrm{False}]\!]_\eta$.

Note that the interpretation map for assertions now also uses two arguments, an environment for first-class variables (values and set types) η and an environment for the predicate variables $\vec{\mathrm{R}}$ and $\vec{\mathrm{I}}$. The interpretation of assertions with extra predicate variables is as in [9]. The only interesting case for interpretation of predicate variables is the following where X denotes either a recursive predicate variable R or an inductive predicate variable $\underline{\mathrm{I}}$.

$$[\![X(\vec{e})]\!]_{\eta,\Xi} \stackrel{def}{=} \begin{cases} \Xi(X)([\![\vec{e}]\!]_\eta) & \text{if } |\vec{e}| = arity(\Xi(X)) \\ false & \text{otherwise} \end{cases} \tag{3}$$

Since we assumed that Ξ contains definitions for all relevant predicate names we do not have to consider the case $\mathrm{X} \notin \mathrm{dom}(\Xi)$. The definition also entails that $[\![X(\vec{e})]\!]_{\eta,\Xi} \in Pred$.

[4] We do not assume a global environment of predicates here as our syntax allows us to define inductive predicates inside recursive ones ("on the fly") and recursive and inductive definitions can make use of other such predicates, respectively, via mutual recursion.

Note also that the functional $\Psi_{\eta,\Xi}$ is uniformly parametric in Ξ and thus the predicate definitions P_i can only refer to predicates $\vec{\underline{\mathrm{I}}}$ that are defined w.r.t. the same parameter instantiations for $\vec{R}$. This is actually enforced by our syntax where inductive calls to any $\underline{\mathrm{I}}_i$ do not mention any predicate parameters $\vec{\mathrm{R}}$ explicitly and thus $\vec{\underline{\mathrm{I}}}(\vec{e})$ has implicit predicate arguments which are $\vec{\mathrm{R}}$ again.

In order to prove the induction rule sound we would like to use that $\Psi_{\eta,\Xi}$ is ω-continuous so let us show this first.

Lemma 5.5. The semantics of formula $P \in \mathscr{B}$ allowed in inductive definitions (as defined in Thm. 4.1) is ω-continuous in its predicate environment, ie. $[\![P]\!]_{\eta,\Xi}$ is continuous in Ξ.

Proof. We have to show that $[\![P]\!]_{\eta,(\sqcup_n \Xi_n)} = \sqcup_n [\![P]\!]_{\eta,\Xi_n}$ where $\sqcup$ on predicates of type *Pred* is pointwise union of the uniform admissible subsets of heaps lifted to environments also in a pointwise manner. This follows easily by induction on the shape of $\mathscr{B}$ using the syntactic restrictions of the grammar for $\mathscr{B}$. As an example, we show the base case:

$$[\![\underline{\mathrm{I}}(\vec{e})]\!]_{\eta,\sqcup_n \Xi_n} = \sqcup_n [\![\underline{\mathrm{I}}(\vec{e})]\!]_{\eta,\Xi_n}$$

We reason as follows:

$$\begin{array}{rcl} [\![\underline{\mathrm{I}}(\vec{e})]\!]_{\eta,\sqcup_n \Xi_n}\ w & = & ((\sqcup_n \Xi_n)(\underline{\mathrm{I}}))([\![\vec{e}]\!]_\eta)\ w \\ & = & \cup_n (\Xi_n(\underline{\mathrm{I}}))([\![\vec{e}]\!]_\eta)\ w \\ & = & \cup_n [\![\underline{\mathrm{I}}(\vec{e})]\!]_{\eta,\Xi_n}\ w \end{array}$$

and because suprema are pointwise on *Pred* we are done. □

Corollary 5.6. The functional $\Psi_{\eta,\Xi}$ that interprets an inductive definition as in (2) is continuous.

Next we show that inductively defined predicates exist and are well behaved.

Lemma 5.7. Each functional $\Psi_{\eta,\Xi}$ that interprets an inductive definition as in (2) has a fixpoint, denoted $\mathit{fix}\ \Psi_{\eta,\Xi}$, that lives in $\mathit{Pred}^{\mathrm{arg}}$.

Proof. By Lemma 5.5, the functional $\Psi_{\eta,\Xi}$, used to interpret an inductive definition, is ω-continuous and thus necessarily monotone. Accordingly, $\mathit{fix}\ \Psi_{\eta,\Xi}$ exists by Tarski's fixpoint theorem. We can assume that all predicates $\Xi(\mathrm{R}_i) \in \mathit{Pred}^{\mathsf{ValS}^m}$ where m is the arity of R_i, that $F(\underline{\mathrm{I}}) \in \mathit{Pred}^{\mathsf{ValS}^{\kappa_i}}$ for all $\underline{\mathrm{I}}$, and need to show that $\mathit{fix}\ \Psi_{\eta,\Xi} \in \mathit{Pred}^{\mathrm{arg}}$. We use the fact that by continuity the following holds:

$$\mathit{fix}\ \Psi_{\eta,\Xi} = \bigsqcup_n \Psi^n_{\eta,\Xi}(\Omega) \tag{4}$$

From this follows immediately for any world w, $k \in \{1,\ldots,n\}$ and $\vec{v} \in \mathsf{ValS}^{\kappa_k}$ that $(\mathit{fix}\ \Psi_{\eta,\Xi})\,(k,\vec{v})\,w$ is uniform and non-expansive in its argument w. Due to the union in the definition of the fixpoint admissibility is not immediate. We need to show that any non-trivial ascending chain in $\bigcup_n \Psi^n_{\eta,\Xi}(\Omega)(k,\vec{v})\,w$ actually

lies within $\Psi^{n_0}_{\eta,\Xi}(\Omega)(k,\vec{v})\,w$ for a particular n_0. This holds for the functionals $\Psi_{\eta,\Xi}$ that interpret inductive definitions of the syntax as given in Thm. 4.1 due to the restriction of $\mathscr{B}$. It is important here that the predicates $\underline{\mathrm{I}}$ must not occur in $\forall\vec{x}.\,\{\Phi_1\}\,e(\vec{e})\,\{\Phi_2\}$. This ensures that application of $\Psi_{\eta,\Xi}(X)$ lets the predicate X grow only "horizontally" adding new heaps that are incomparable to the ones in X, but not "vertically", thus avoiding the creation of chains. □

The fixpoint can now be used to interpret inductive predicates as they appear in assertions:

$$[\![(\mu^k_{ind}\left\langle\vec{\mathrm{R}}\right\rangle\ \underline{\mathrm{I}}_1(\vec{x_1}),\ldots,\underline{\mathrm{I}}_n(\vec{x_n}).\ P_1,\ldots,P_n)(\mathrm{S}_1,\ldots\mathrm{S}_n)(\vec{e})]\!]_{\eta,\Xi} \stackrel{def}{=}$$

$$\begin{cases} (\mathit{fix}\,\Psi_{\eta,[\mathrm{R}_i\mapsto[\![\mathrm{S}_i]\!]_\Xi]})(k,[\![\vec{e}]\!]_\eta) & \text{if } [\![\vec{e}]\!]_\eta \in \mathsf{ValS}^{\kappa_k} \\ \mathit{false} & \text{otherwise} \end{cases} \tag{5}$$

Next, we need the following lemma:

Lemma 5.8. The interpretation of any elimination formula Φ with free predicate variable X is admissible in the following sense:

$$(\forall n.\,\forall\eta.\,[\![\Phi]\!]_{\eta,[\mathrm{X}\mapsto A_n]} = [\![\mathrm{True}]\!]_\eta) \ \Rightarrow\ \forall\eta.\,[\![\Phi]\!]_{\eta,[\mathrm{X}\mapsto\sqcup_n A_n]} = [\![\mathrm{True}]\!]_\eta \tag{6}$$

where $(A_n)_{n\in\mathbb{N}}$ is an ascending chain in $\mathit{Pred}^{\mathsf{ValS}^\kappa}$ for some arity κ.

Proof. This is easily shown using the fact that *elimination formula* Φ is of the shape $\forall\vec{x}.\,\mathrm{X}(\vec{e}) \star \Phi_0 \Rightarrow \Phi_1$. One uses the fact that if $h \in \bigcup_n A_n([\![\vec{e}]\!]_\eta)\,w \cdot [\![\Phi_0]\!]_\eta\,w$ then there must be a n_0 such that $h \in A_{n_0}([\![\vec{e}]\!]_\eta)\,w \cdot [\![\Phi_0]\!]_\eta\,w$ and thus by assumption one gets the desired $[\![\Phi_1]\!]_\eta\,w$. □

Theorem 5.9. (MuInd) and (Induction) in Fig. 11 hold.

Proof. Soundness of (MuInd) is a consequence of equations (2), (3), and (5) above, making use of the fixpoint property. Soundness of (Induction) uses equations (4) and (6) and is shown as follows.

Let $\Psi_{\eta,\Xi}$ be the semantic functional the inductive definition gives rise to. We know that $\Psi_{\eta,\Xi[\mathrm{X}\mapsto Z]}$ is ω-continuous in Z (Lemma 5.5), so by equation (4) it suffices to show

$$[\![\Phi]\!]_{\eta,[\mathrm{X}\mapsto(\bigsqcup_n \Psi^n_{\eta,\Xi})(\Omega)]} = [\![\mathrm{True}]\!]_\eta$$

where Ξ is the environment for $\vec{R}$ parameters and defined predicates in use. We know that Φ is admissible so by (6) it suffices to show

$$\forall n.\,[\![\Phi]\!]_{\eta,[\mathrm{X}\mapsto\Psi^n_{\eta,\Xi}(\Omega)]} = [\![\mathrm{True}]\!]_\eta$$

This is shown by induction on n. The two premises of (Induction) correspond to the induction base case and induction step, respectively. For $n = 0$ we need to show $[\![\Phi]\!]_{\eta,[\mathrm{X}\mapsto\lambda\vec{x}.\,[\![\mathrm{False}]\!]_\eta]} = [\![\mathrm{True}]\!]_\eta$ which follows from the appropriate substitution lemma from $[\![\Phi[X \mapsto \lambda\vec{x}.\,\mathrm{False}]]\!]_\eta = [\![\mathrm{True}]\!]_\eta$ which is the semantics of the first assumption of the rule.

For the induction step we need to show for any $n \in \mathbb{N}$ that

$$[\![\Phi]\!]_{\eta,[\mathrm{X}\mapsto\Psi^n_{\eta,\Xi}(\Omega)]} = [\![\mathrm{True}]\!]_\eta \quad \Rightarrow \quad [\![\Phi]\!]_{\eta,[\mathrm{X}\mapsto\Psi^{n+1}_{\eta,\Xi}(\Omega)]} = [\![\mathrm{True}]\!]_\eta \tag{7}$$

But now we observe that

$$\begin{array}{lcl}
[\![\Phi]\!]_{\eta,[\mathrm{X}\mapsto\Psi^{n+1}_{\eta,\Xi}(\Omega)]} & = & \\
[\![\Phi]\!]_{\eta,[\mathrm{X}\mapsto\Psi_{\eta,\Xi}(\Psi^n_{\eta,\Xi}(\Omega))]} & = & \text{Definition of } \Psi_{\eta,\Xi} \\
[\![\Phi]\!]_{\eta,[\mathrm{X}\mapsto[\![P]\!]_{\eta,\Xi[\underline{I}\mapsto\Psi^n_{\eta,\Xi}(\Omega),\mathrm{R}_i\mapsto[\![\mathrm{S}_i]\!]_\eta]}]} & = & \text{Subst.Lem. and } \vec{R} \notin fv(\vec{\mathrm{S}},\Phi) \\
[\![\Phi]\!]_{\eta,[\mathrm{X}\mapsto[\![P[\underline{\mathrm{I}},\vec{\mathrm{R}}\backslash\mathrm{X},\vec{\mathrm{S}}]]\!]_{\eta,[\mathrm{X}\mapsto\Psi^n_{\eta,\Xi}(\Omega)]}]} & = & \\
[\![\Phi[\mathrm{X}\backslash P[\underline{\mathrm{I}},\vec{\mathrm{R}}\backslash\mathrm{X},\vec{\mathrm{S}}]]]\!]_{\eta,[\mathrm{X}\mapsto\Psi^n_{\eta,\Xi}(\Omega)]} & &
\end{array}$$

Note that the domain of Ξ may contain $\underline{\mathrm{I}}$ if $\underline{\mathrm{I}}$ occurs in Φ. In this case $\underline{\mathrm{I}}$ is only temporarily overwritten in the environment for P in order to provide the correct semantics of P. Due to the above equation, condition (7) is equivalent to

$$[\![\Phi]\!]_{\eta,[\mathrm{X}\mapsto\Psi^n_{\eta,\Xi}(\Omega)]} = [\![\mathrm{True}]\!]_\eta \quad \Rightarrow \quad [\![\Phi[\mathrm{X}\backslash P[\underline{\mathrm{I}},\vec{\mathrm{R}}\backslash\mathrm{X},\vec{\mathrm{S}}]]]\!]_{\eta,[\mathrm{X}\mapsto\Psi^n_{\eta,\Xi}(\Omega)]} = [\![\mathrm{True}]\!]_\eta$$

which follows from

$$[\![\Phi]\!]_\eta = [\![\mathrm{True}]\!]_\eta \quad \Rightarrow \quad [\![\Phi[\mathrm{X}\backslash P[\underline{\mathrm{I}},\vec{\mathrm{R}}\backslash\mathrm{X},\vec{\mathrm{S}}]]]\!]_\eta = [\![\mathrm{True}]\!]_\eta$$

which in turn follows from[5]

$$\forall\eta.\, [\![\Phi \quad \Rightarrow \quad \Phi[\mathrm{X}\backslash P[\underline{\mathrm{I}},\vec{\mathrm{R}}\backslash\mathrm{X},\vec{\mathrm{S}}]]]\!]_\eta = [\![\mathrm{True}]\!]_\eta$$

which is the semantics of the second hypothesis of the (INDUCTION) rule, so we are done. □

5.5. Existence of recursive assertions (Soundness of Theorem 4.1)

We show the existence of the most general mutually recursively defined predicate pattern (the "master pattern") from Theorem 4.1.

We need to define $[\![(\mu^k\ \mathrm{R}_1(\vec{x}_1),\ldots,\mathrm{R}_n(\vec{x}_n)\ .\ P_1,\ldots,P_n)(\vec{e})]\!]_\eta$ but also the meaning of recursive predicate expressions without arguments needs to be defined, $[\![(\mu^k\ \mathrm{R}_1(\vec{x}_1),\ldots,\mathrm{R}_n(\vec{x}_n)\ .\ P_1,\ldots,P_n)]\!]_\eta$, as they can be used as arguments themselves.

The interpretation does not depend on a predicate environment as any other inductive predicate used can be defined "on the fly" and other recursive predicates can be used by mutual recursion. We re-use the definition of type arg from the previous subsection but now κ_i refers to the arity of the declared predicate names R_i. We can then define a functional $\Psi_\eta : \mathit{Pred}^{\mathrm{arg}} \to \mathit{Pred}^{\mathrm{arg}}$ as follows:

$$\Psi_\eta(F)(k,\vec{v}) \stackrel{def}{=} [\![P_k]\!]_{\eta[\vec{x_k}\mapsto\vec{v}],\,[\mathrm{R}_i\mapsto\lambda\vec{y}_i.F(i,\vec{y}_i)]} \tag{8}$$

[5] Note that in our model implication receives a non-standard interpretation which is uniform in the worlds (see [9, 12]) but since it is stronger than the one used here this is fine.

where $|\vec{v}| = \kappa_k$ is the arity of R_k and $|\vec{y}_i| = \kappa_i$ is the arity of R_i.

We will now argue below that Ψ_η as defined above has a fixpoint $\mathit{fix}\,\Psi_\eta$ in which case one can define in analogy to the inductive case:

$$[\![\mu^k\ \mathrm{R}_1(\vec{x}_1), \ldots, \mathrm{R}_n(\vec{x}_n).\ P_1, \ldots, P_n]\!]_\eta \stackrel{def}{=} \mathit{fix}\,\Psi_\eta \tag{9}$$

$$[\![(\mu^k\ \mathrm{R}_1(\vec{x}_1), \ldots, \mathrm{R}_n(\vec{x}_n).\ \vec{P}_i)(\vec{e})]\!]_\eta \stackrel{def}{=} \begin{cases} (\mathit{fix}\,\Psi_\eta)(k, [\![\vec{e}]\!]_\eta) & \text{if } [\![\vec{e}]\!]_\eta \in \mathsf{ValS}^{\kappa_k} \\ \mathit{false} & \text{otherwise} \end{cases}$$

Proof of Theorem 4.1. We actually show that the functional $\Psi_\eta : \mathit{Pred}^{\arg} \to \mathit{Pred}^{\arg}$ to interpret $\mu\ \mathrm{R}_1(\vec{x}_1), \ldots, \mathrm{R}_n(\vec{x}_n).\, P_1, \ldots, P_n$ (defined in (8)) is contractive and thus has a fixpoint by Banach's fixpoint theorem [21]. From [9, Thm. 12] we already know that it suffices to show that for all $(i, v) \in \arg$ we have $\Psi^* : \mathit{Pred}^{\arg} \to \mathit{Pred}$ where $\Psi^*(F) = \Psi_\eta(F)(i, v)$ is a contractive function in F. According to the allowable patterns $\mathscr{A}$, as defined in Thm. 4.1, we can show by induction on the structure of $\mathscr{A}$ that this is the case. Only two cases actually depend on the recursive predicates. In the first one, $\{\Phi\}\, e(\vec{e})\, \{\Phi\}$, the R_i can appear in formulae Φ but the semantics of triples as defined in [9, 12] is carefully constructed such that $[\![\{\Phi\}\, e(\vec{e})\, \{\Phi\}]\!]$ is contractive in $\vec{\mathrm{R}}$. This has been shown in [12, Lemma 31]. The second case of interest is $I(\vec{\mathrm{R}})(\vec{e})$ where $I \in \mathfrak{I}$, so we have to consider assertions of the form

$$(\mu^k_{ind} \left\langle \vec{\mathrm{S}} \right\rangle\ \underline{\mathrm{I}}_1(\vec{x_1}), \ldots, \underline{\mathrm{I}}_n(\vec{x_n}).\ P_1, \ldots, P_n)(\vec{\mathrm{R}})(\vec{e})$$

where all $P_i \in \mathscr{B}$. But $\mathscr{B}$ restricts occurrences of the parameters $\vec{\mathrm{S}}$ to appear within Φ's inside $\{\Phi\}\, e(\vec{e})\, \{\Phi\}$ which again establishes contractiveness in $\vec{\mathrm{R}}$ (as in the previous case) since actual parameters $\vec{\mathrm{R}}$ are substituted for formal parameters $\vec{\mathrm{S}}$. We also need to show that all other logical connectives, like emp, $\_ * \_$ and $\_ \wedge \_$, are non-expansive in each argument, but this is straightforward (see [9, 12]). □

The intuition behind the contractiveness of our master pattern is that recursive parameters R must appear inside nested triples. The interpretation of those (not mentioned here but to be found in [9, 12]) is defined in a way that enforces contractiveness. Heaps that satisfy the pre- or post-condition can be viewed as "once unfolded according to their recursive definition" justifying the increase in distance.

Again, any of the recursively defined predicates are admissible and uniform since fixpoints of contractive maps between uniform admissible predicates are automatically uniform and admissible (see [22])

Theorem 5.10. (Mu) in Fig. 11 holds.

Proof. Soundness of (Mu) is a consequence of the semantics above, see equations (8) and (9), making use of the fixpoint property. □

6. Specifying and proving our example program

In this section we show how to specify memory safety of our running example program. Because proofs about programs which recurse through the store can look a little odd to the untrained eye, we introduce them by first reasoning about the tree disposal code DT. We will then go on to prove safety of the main part of the program.

6.1. Proving correctness of the tree disposal code

We have already described, on page 14, the property we would like to prove of our DT code; it is

$$\forall t. \left\{ \begin{array}{l} \exists \tau.\, \mathrm{tree}(t, \tau) \\ \star\ \mathit{DisposeTreeCode} \end{array} \right\}\ \text{`}\lambda\ \mathit{tree}.\ DT_{\mathrm{body}}\text{'}(t)\ \ \{\ \mathit{DisposeTreeCode}\ \}$$

where *DisposeTreeCode* is shorthand for

$$\mu^1\ \mathrm{R}\ .\ \mathit{disposeTree} \mapsto \forall t.\, \{\exists \tau.\mathrm{tree}(t, \tau) \star \mathrm{R}\} \cdot (t)\, \{\mathrm{R}\}$$

and DT_{body} is the body of the DT code. Note that *DisposeTreeCode* is equivalent to

$$\mu^1\ \mathrm{R}\ .\ \mathit{disposeTree} \mapsto \forall t.\, \{\exists \tau.\mathrm{tree}(t, \tau)\} \cdot (t)\, \{\mathsf{emp}\} \otimes \mathrm{R}$$

The proof makes prominent use of the (Eval) rule, for reasoning about calls to stored code; the (Mu) rule, for folding and unfolding the recursively defined predicate *DisposeTreeCode*, and the (Lambda) rule for dealing with parameter passing.

We begin by applying the (Lambda) rule to deal with the parameter passing, leaving us to prove

$$\left\{ \begin{array}{l} \exists \tau.\mathrm{tree}(\mathit{tree}, \tau) \\ \star\ \mathit{DisposeTreeCode} \end{array} \right\}\ DT_{\mathrm{body}}\ \ \{\ \mathit{DisposeTreeCode}\ \}$$

We now unfold $\mathrm{tree}(\mathit{tree}, \tau)$ (using (MuInd)) in the precondition which becomes a disjunction of two cases, the leaf case and the fork case. The leaf case is straightforward so we concentrate on the fork case, where need to prove:

$$\left\{ \begin{array}{l} \exists \tau, \mathit{cPtr}, \mathit{left}, \mathit{right}, \mathit{opId}, \tau', \tau''. \\ \quad \mathit{tree} \mapsto \mathrm{opLbl}, \mathit{cPtr}, l, r, \mathit{opId} \\ \star\ \mathrm{tree}(\mathit{left}, \tau') \\ \star\ \mathrm{tree}(\mathit{right}, \tau'') \\ \star\ \mathit{DisposeTreeCode} \\ \wedge\ \tau = \{\mathit{cPtr}\} \cup \tau' \cup \tau'' \end{array} \right\}\ DT_{\mathrm{body}}\ \ \{\ \mathit{DisposeTreeCode}\ \}$$

Dealing with the let and then the if statement, which must enter the else branch, we are left with:

$$\left\{\begin{array}{l}\exists \tau, cPtr, l, r, opId, \tau', \tau''. \\ \quad tree \mapsto \text{opLbl}, cPtr, l, r, opId \\ \star \text{tree}(l, \tau') \\ \star \text{tree}(r, \tau'') \\ \star \mathit{DisposeTreeCode} \\ \wedge\, \tau = \{cPtr\} \cup \tau' \cup \tau'' \\ \wedge\, kind = \text{opLbl}\end{array}\right\} \begin{array}{l}\mathsf{let}\ left = [tree + \text{leftO}]\ \mathsf{in} \\ \mathsf{let}\ right = [tree + \text{rightO}] \\ \ \mathsf{in} \\ \mathsf{free}\ tree\ ; \\ \mathsf{free}\ tree + \text{CodePtrO}\ ; \\ \mathsf{free}\ tree + \text{LeftO}\ ; \\ \mathsf{free}\ tree + \text{RightO}\ ; \\ \mathsf{free}\ tree + \text{OpIDO}\ ; \\ \mathsf{eval}\ [disposeTree](left)\ ; \\ \mathsf{eval}\ [disposeTree](right)\end{array} \quad \{\ \mathit{DisposeTreeCode}\ \}$$

This easily reduces to

$$\left\{\begin{array}{l}\exists \tau, codePtr, \tau', \tau''. \\ \quad \text{tree}(left, \tau') \\ \star \text{tree}(right, \tau'') \\ \star \mathit{DisposeTreeCode} \\ \wedge\, \tau = \{codePtr\} \cup \tau' \cup \tau'' \\ \wedge\, kind = \text{opLbl}\end{array}\right\} \begin{array}{l}\mathsf{eval}\ [disposeTree](left)\ ; \\ \mathsf{eval}\ [disposeTree](right)\end{array} \quad \{\ \mathit{DisposeTreeCode}\ \}$$

This leads us to the more interesting reasoning steps. By a combination of (ForallExists), universal generalisation and the consequence rule, it will be enough to prove:

$$\left\{\begin{array}{l}\text{tree}(left, \tau') \\ \star \text{tree}(right, \tau'') \\ \star \mathit{DisposeTreeCode}\end{array}\right\} \begin{array}{l}\mathsf{eval}\ [disposeTree](left)\ ; \\ \mathsf{eval}\ [disposeTree](right)\end{array} \quad \{\ \mathit{DisposeTreeCode}\ \}$$

We break this down by sequential composition into

$$\left\{\begin{array}{l}\text{tree}(left, \tau') \\ \star \text{tree}(right, \tau'') \\ \star \mathit{DisposeTreeCode}\end{array}\right\} \ \mathsf{eval}\ [disposeTree](left) \ \left\{\begin{array}{l}\text{tree}(right, \tau'') \\ \star \mathit{DisposeTreeCode}\end{array}\right\} \tag{10}$$

and

$$\left\{\begin{array}{l}\text{tree}(right, \tau'') \\ \star \mathit{DisposeTreeCode}\end{array}\right\} \ \mathsf{eval}\ [disposeTree](right) \ \{\ \mathit{DisposeTreeCode}\ \} \tag{11}$$

Using the frame rule ($\star$-Frame), taking $\text{tree}(right, \tau'')$ as the framed invariant, we can reduce (10) to a case which is symmetric to (11). Thus we will only prove (11). We begin by using the (Mu) rule to unfold the use of *DisposeTreeCode* in the precondition, leaving us with

$$\left\{\begin{array}{l}\text{tree}(right, \tau'') \\ \star\ dispose Tree \mapsto \forall t. \\ \qquad \left\{\begin{array}{l}\exists \tau. \text{tree}(t, \tau) \\ \star \mathit{DisposeTreeCode}\end{array}\right\} \\ \qquad\qquad \cdot(t) \\ \qquad \{\ \mathit{DisposeTreeCode}\ \}\end{array}\right\} \ \mathsf{eval}\ [disposeTree](right) \ \{\ \mathit{DisposeTreeCode}\ \}$$

Then, using the (EVAL) rule, we will be finished if we can show

$$\forall t.\left\{\begin{array}{l}\exists\tau.\mathrm{tree}(t,\tau)\\ \star\ \mathit{DisposeTreeCode}\end{array}\right\}\ k(t)\ \{\ \mathit{DisposeTreeCode}\ \}$$

$$\Rightarrow$$

$$\left\{\begin{array}{l}\mathrm{tree}(\mathit{right},\tau'')\\ \star\ \mathit{disposeTree}\mapsto\forall t.\\ \quad\left\{\begin{array}{l}\exists\tau.\mathrm{tree}(t,\tau)\\ \star\ \mathit{DisposeTreeCode}\end{array}\right\}\\ \qquad\cdot(t)\\ \quad\{\ \mathit{DisposeTreeCode}\ \}\end{array}\right\}\ k(\mathit{right})\ \{\ \mathit{DisposeTreeCode}\ \}$$

To prove this we argue as follows.

$$\forall t.\left\{\begin{array}{l}\exists\tau.\mathrm{tree}(t,\tau)\\ \star\ \mathit{DisposeTreeCode}\end{array}\right\}\ k(t)\ \{\ \mathit{DisposeTreeCode}\ \}$$

$\Rightarrow$ {instantiate t with right}

$$\left\{\begin{array}{l}\exists\tau.\mathrm{tree}(\mathit{right},\tau)\\ \star\ \mathit{DisposeTreeCode}\end{array}\right\}\ k(\mathit{right})\ \{\ \mathit{DisposeTreeCode}\ \}$$

$\Rightarrow$ {use (MU) to unfold $\mathit{DisposeTreeCode}$ in the precondition}

$$\left\{\begin{array}{l}\exists\tau.\mathrm{tree}(\mathit{right},\tau)\\ \star\ \mathit{disposeTree}\mapsto\forall t.\\ \quad\left\{\begin{array}{l}\exists\tau.\mathrm{tree}(t,\tau)\\ \star\ \mathit{DisposeTreeCode}\end{array}\right\}\\ \qquad\cdot(t)\\ \quad\{\ \mathit{DisposeTreeCode}\ \}\end{array}\right\}\ k(\mathit{right})\ \{\ \mathit{DisposeTreeCode}\ \}$$

$\Rightarrow$ {consequence rule}

$$\left\{\begin{array}{l}\mathrm{tree}(\mathit{right},\tau'')\\ \star\ \mathit{disposeTree}\mapsto\forall t.\\ \quad\left\{\begin{array}{l}\exists\tau.\mathrm{tree}(t,\tau)\\ \star\ \mathit{DisposeTreeCode}\end{array}\right\}\\ \qquad\cdot(t)\\ \quad\{\ \mathit{DisposeTreeCode}\ \}\end{array}\right\}\ k(\mathit{right})\ \{\ \mathit{DisposeTreeCode}\ \}$$

With this relatively simple proof under our belt, let us tackle the specification and proof of the main part of our program.

6.2. Proof of safety of the main program

Our program uses three heap data structures (Fig. 1) — an expression tree, an association list (mapping variables to values) and a list of code — as well as various procedures stored on the heap. Fig. 12 defines the predicates we use to describe these; all uses of μ fit the master pattern. Fig. 12 first defines *AssocList*

$$AssocList \quad := \quad \exists \sigma, a \;.\; assoclist \mapsto a \star \mathrm{lseg}(a, \mathsf{null}, \sigma)$$

$$(CLseg, EvalCode, LoaderCode, SearchModsCode) := \mu\, \mathrm{CLseg}(x, y, \sigma), \mathrm{EC}, \mathrm{LC}, \mathrm{SMC} \;.$$

$$\mathrm{lseg}[\vec{\mathrm{R}}, Mod(\cdot)][\mathrm{CLseg}, \mathrm{EC}, \mathrm{LC}, \mathrm{SMC}](x, y, \sigma),$$

$$evalTree \mapsto \forall t, r, \tau_1, \sigma_1 \;.$$
$$\left\{ \begin{array}{l} \exists a. \quad \tau_1 \subseteq \sigma_1 \wedge \\ \quad\; modlist \mapsto a \\ \star\, \mathrm{CLseg}(a, \mathsf{null}, \sigma_1) \\ \star\, \mathrm{tree}(t, \tau_1) \\ \star\, \mathrm{EC} \\ \star\, \mathrm{LC} \\ \star\, \mathrm{SMC} \\ \star\, r \mapsto \_ \end{array} \right\} \cdot(t, r) \left\{ \begin{array}{l} \exists a, \sigma_2, \tau_2. \quad \tau_2 \subseteq \sigma_2 \wedge \sigma_1 \subseteq \sigma_2 \wedge \\ \quad\; modlist \mapsto a \\ \star\, \mathrm{CLseg}(a, \mathsf{null}, \sigma_2) \\ \star\, \mathrm{tree}(t, \tau_2) \\ \star\, \mathrm{EC} \\ \star\, \mathrm{LC} \\ \star\, \mathrm{SMC} \\ \star\, r \mapsto \_ \end{array} \right\},$$

$$loader \mapsto \forall opID, codePtraddr, \sigma_1 \;.$$
$$\left\{ \begin{array}{l} \exists a \;. \\ \quad\; modlist \mapsto a \\ \star\, \mathrm{CLseg}(a, \mathsf{null}, \sigma_1) \\ \star\, codePtraddr \mapsto \_ \end{array} \right\} \cdot(opID, codePtraddr) \left\{ \begin{array}{l} \exists a, r, \sigma_2. \; \sigma_1 \cup \{r\} \subseteq \sigma_2 \\ \wedge\; modlist \mapsto a \\ \star\, \mathrm{CLseg}(a, \mathsf{null}, \sigma_2) \\ \star\, codePtraddr \mapsto r \end{array} \right\},$$

$$searchMods \mapsto \forall opID, codePtraddr, a, \sigma \;.$$
$$\left\{ \begin{array}{l} \quad\; modlist \mapsto a \\ \star\, \mathrm{CLseg}(a, \mathsf{null}, \sigma) \\ \star\, codePtraddr \mapsto \_ \end{array} \right\} \cdot (opID, codePtraddr) \left\{ \begin{array}{l} \exists r. \quad (r \in \sigma \vee r = \mathsf{null}) \\ \wedge\; modlist \mapsto a \\ \star\, \mathrm{CLseg}(a, \mathsf{null}, \sigma) \\ \star\, codePtraddr \mapsto r \end{array} \right\}$$

where $\vec{\mathrm{R}}$ is $\mathrm{CLseg}(\cdot, \cdot, \cdot), \mathrm{EC}, \mathrm{LC}, \mathrm{SMC}$ and $Mod(\cdot)$ is

$$\forall t, r, \tau_1, \sigma_1 \;.$$
$$\left\{ \begin{array}{l} \exists a. \quad \tau_1 \subseteq \sigma_1 \wedge \\ \quad\; modlist \mapsto a \\ \star\, \mathrm{CLseg}(a, \mathsf{null}, \sigma_1) \\ \star\, \mathrm{tree}_{\mathrm{fork}}(t, \tau_1) \\ \star\, \mathrm{EC} \\ \star\, \mathrm{LC} \\ \star\, \mathrm{SMC} \\ \star\, r \mapsto \_ \end{array} \right\} \cdot(t, r) \left\{ \begin{array}{l} \exists a, \sigma_2, \tau_2. \quad \tau_2 \subseteq \sigma_2 \wedge \sigma_1 \subseteq \sigma_2 \wedge \\ \quad\; modlist \mapsto a \\ \star\, \mathrm{CLseg}(a, \mathsf{null}, \sigma_2) \\ \star\, \mathrm{tree}(t, \tau_2) \\ \star\, \mathrm{EC} \\ \star\, \mathrm{LC} \\ \star\, \mathrm{SMC} \\ \star\, r \mapsto \_ \end{array} \right\}$$

$$Mod'(F) :=$$
$$Mod(F)[\mathrm{CLseg}, \mathrm{EC}, \mathrm{LC}, \mathrm{SMC} \backslash CLseg, EvalCode, LoaderCode, SearchModsCode]$$

Figure 12: Definitions of predicates used in the specification of our example program.

to describe the association list. Then we define, in one mutually recursive definition, four predicates *CLseg*, *EvalCode*, *LoaderCode* and *SearchModsCode* for describing the code list and the procedures stored on the heap.

Predicate $CLseg(x, z, \sigma)$ denotes a segment of the code list, running from address x to z, where σ is the set of addresses of the list nodes (and thus, of the modules) in the list. Predicate *EvalCode* denotes a procedure stored at address *evalTree* on the heap and satisfying a specification expressing the required behaviour of a safe tree evaluation routine. Similarly *LoaderCode* and *SearchModsCode* describe appropriately behaved dynamic loading and module list search procedures stored on the heap at addresses *loader* and *searchMods* respectively.

The abbreviation $Mod(F)$ used in the recursive definition says that code F behaves appropriately to be used as a module in our system. $Mod(F)$ is almost the same as the specification used for *evalTree* (which we shall call *EvalTriple*); the difference is that modules may assume the tree is a fork, because *evalTree* has already checked for and handled the leaf case. Mod' is the same as *Mod* but with the recursion variables CLseg, EC, LC, SMC replaced by the (top-level) predicates *CLseg*, *EvalCode*, *LoaderCode*, *SearchModsCode*. Thus we can use Mod' at the top level of our proofs.

The code list segment predicate *CLseg* satisfies standard properties of list segment predicates, such as the implication

$$CLseg(x, z, \sigma) \wedge a \in \sigma \quad \Rightarrow \quad \exists y, \sigma_1, \sigma_2. \left(\begin{array}{c} CLseg(x, a, \sigma_1) \\ \star\ a \mapsto Mod'(\cdot), \_, y \\ \star\ CLseg(y, z, \sigma_2) \\ \wedge\ \sigma = \sigma_1 \cup \{a\} \cup \sigma_2 \end{array} \right) \tag{12}$$

which allows one to split the list node at address a out from the middle of the list. Such properties are proved using (INDUCTION) and we will use them in our proofs.

We will prove the main program (Fig. 2) satisfies the following specification:

$$\left\{ \begin{array}{l} \mathit{modlist} \mapsto a \star CLseg(a, \mathsf{null}, \sigma) \\ \star\ \mathrm{tree}(\mathit{tree}, \tau) \wedge \tau \subseteq \sigma \\ \star\ \mathit{LoaderCode} \star \mathit{SearchModsCode} \\ \star\ \mathit{evalTree} \mapsto \_ \end{array} \right\} \ \text{MainProg} \ \{ \ \text{True} \ \} \tag{13}$$

A more specific post-condition could be used to prove the absence of memory leaks etc., but True is sufficient for memory safety. Note that we *assume* specifications for the procedures stored at *loader* and *searchMods* (for which the code is not given). On the other hand we *prove* (as a sub-proof) that the code which the main program writes to *evalTree* satisfies the specification in the *EvalCode* predicate.

The reader may wonder why the triple (13) makes no mention of the association list. Shouldn't this appear in the precondition, since the modules invoked by the program may access it? In fact, we have not mentioned *AssocList* anywhere in the recursive definition in Fig. 12. Our idea is to prove safety of the

main program as if the association list was not used, and afterwards use the deep frame rule to add it everywhere it is needed. This keeps the proof simpler, and is possible because the main program doesn't manipulate the association list directly, only via the loadable modules. Specifically, we use ($\star$-FRAME) to add *AssocList* as an invariant to (13); using the distribution laws for $\otimes$, this gives us

$$\left\{\begin{array}{l} \quad \mathit{modlist} \mapsto a \\ \star\ \mathit{CLseg}(a, \mathsf{null}, \sigma) \otimes \mathit{AssocList} \\ \star\ \mathrm{tree}(\mathit{tree}, \tau) \wedge \tau \subseteq \sigma \\ \star\ \mathit{LoaderCode} \otimes \mathit{AssocList} \\ \star\ \mathit{SearchModsCode} \otimes \mathit{AssocList} \\ \star\ \mathit{evalTree} \mapsto \_ \\ \star\ \mathit{AssocList} \end{array}\right\}\ \mathrm{MainProg}\ \left\{\begin{array}{c} \mathrm{True} \\ \star\ \mathit{AssocList} \end{array}\right\}$$

Using the consequence rule we can now weaken the postcondition to True for simplicity.

The full proof of (13) is rather large, so here we only sketch the part of the proof, namely the following triple:

$$\forall t, r, \sigma_1, \tau_1\ .\ \left\{\begin{array}{l} \exists a\ . \\ \quad \mathit{modlist} \mapsto a \\ \star\ \mathit{CLseg}(a, \mathsf{null}, \sigma_1) \\ \star\ \mathrm{tree}(t, \tau_1) \\ \star\ \mathit{EvalCode} \\ \star\ \mathit{LoaderCode} \\ \star\ \mathit{SearchModsCode} \\ \star\ r \mapsto \_ \\ \wedge\ \tau_1 \subseteq \sigma_1 \end{array}\right\}$$

$$\text{`}\lambda\ \mathit{tree}, \mathit{resaddr}.\ \mathsf{let}\ \mathit{kind} = [\mathit{tree}]\ \mathsf{in}\ \mathsf{if}\ \mathit{kind} = \mathrm{leafLabel}\ \mathsf{then}\ [\mathit{resaddr}] := [\mathit{tree} + \mathrm{ValO}]\ \mathsf{else}\ \mathsf{let}\ \mathit{codeaddr} = [\mathit{tree} + \mathrm{CodePtrO}]\ \mathsf{in}\ \mathsf{eval}\ [\mathit{codeaddr}]\ (\mathit{tree}, \mathit{resaddr})\text{'}\ (t, r)$$

$$\left\{\begin{array}{l} \exists a, \sigma_2, \tau_2\ . \\ \quad \mathit{modlist} \mapsto a \\ \star\ \mathit{CLseg}(a, \mathsf{null}, \sigma_2) \\ \star\ \mathrm{tree}(t, \tau_2) \\ \star\ \mathit{EvalCode} \\ \star\ \mathit{LoaderCode} \\ \star\ \mathit{SearchModsCode} \\ \star\ r \mapsto \_ \\ \wedge\ \tau_2 \subseteq \sigma_2 \\ \wedge\ \sigma_1 \subseteq \sigma_2 \end{array}\right\}$$

which is the proof obligation resulting from writing the code above into *evalTree* stating that it has the required behaviour. The proof proceeds by using the (LAMBDA) rule and then unfolding the tree predicate (with (MUIND)), which results in a disjunction of two cases, namely when the tree is a fork and when it is a leaf. The leaf case is easy so we omit it, concentrating on the fork case where recursion through the store happens. Standard reasoning for the "let" and "if-then-else" constructs, as well as list reasoning for the *CLseg* predicate (using (12) to expose the cell containing the code we will call) reduces the proof obligation to the following:

$$\left\{\begin{array}{l}\exists a, l, r, \tau_{\mathrm{l}}, \tau_{\mathrm{r}}, \\ \qquad nxt, \sigma_{\mathrm{pref}}, \sigma_{\mathrm{suff}} \ . \\ \quad modlist \mapsto a \\ \star\ CLseg(a, codeaddr, \sigma_{\mathrm{pref}}) \\ \star\ codeaddr \mapsto Mod(\cdot), \_, nxt \\ \star\ CLseg(nxt, \mathsf{null}, \sigma_{\mathrm{suff}}) \\ \star\ tree \mapsto \mathrm{opLbl}, codeaddr, l, r, \_ \\ \star\ \mathrm{tree}(l, \tau_{\mathrm{l}}) \\ \star\ \mathrm{tree}(r, \tau_{\mathrm{r}}) \\ \star\ EvalCode \\ \star\ LoaderCode \\ \star\ SearchModsCode \\ \star\ resaddr \mapsto \_ \\ \wedge\ \tau_1 \subseteq \sigma_1 \\ \wedge\ \tau_1 = \{codeaddr\} \cup \tau_{\mathrm{l}} \cup \tau_{\mathrm{r}} \\ \wedge\ \sigma_1 = \sigma_{\mathrm{pref}} \cup \{codeaddr\} \cup \sigma_{\mathrm{suff}} \end{array}\right\} \begin{array}{l}\text{`}\mathsf{eval}\ [codeaddr] \\ (tree, resaddr)\text{'}\end{array} \left\{\begin{array}{l}\exists a, \sigma_2, \tau_2 \ . \\ \quad modlist \mapsto a \\ \star\ CLseg(a, \mathsf{null}, \sigma_2) \\ \star\ \mathrm{tree}(tree, \tau_2) \\ \star\ EvalCode \\ \star\ LoaderCode \\ \star\ SearchModsCode \\ \star\ resaddr \mapsto \_ \\ \wedge\ \tau_2 \subseteq \sigma_2 \\ \wedge\ \sigma_1 \subseteq \sigma_2 \end{array}\right\} \tag{14}$$

Now we can finally apply the (EVAL) rule; the implication we need to prove is

$$Mod'(k) \Rightarrow \Psi$$

where Ψ is the target triple (14) with the body replaced by $k(tree, resaddr)$. The implication can be shown using the consequence rule, folding for the $\mathrm{tree}_{\mathrm{fork}}$ and *CLseg* definitions, and the definition of Mod'.

6.3. Proving safety of the loadable modules

To prove memory safety of the modules we must show for each module implementation M that $Mod'(M) \otimes AssocList$. For modules that do not directly access the association list (e.g. PLUS), we first prove $Mod'(M)$ and then use the deep frame rule $\otimes$–FRAME to add *AssocList*. Appendix A contains the proof for the PLUS module. In that proof one sees the purpose of the constraint $\sigma_1 \subseteq \sigma_2$ in the postconditions of *Mod* and *EvalTriple*, which says that modules can update code in place, and add new code, but may not *delete* nodes from the code list. If modules were deleted while PLUS was evaluating the left subtree, pointers to that code might still remain in the right subtree, leading to a crash.

For modules that *do* access the association list, e.g. VAR and ASSIGN, this cannot be done; one must prove $Mod(M) \otimes AssocList$ directly. The distribution axioms for $\otimes$ play a key role in doing this. For purposes of illustration, consider a module INCR_ALL whose job is to (ignore its arguments and) increment the value of every variable in the association list. This module is implemented by a command C satisfying

$$\forall t, r.\, \{AssocList\}\, C(t, r)\, \{AssocList\} \tag{15}$$

because the association list is the only thing INCR_ALL needs to access. We now sketch how to derive the required $Mod'(C) \otimes AssocList$ from (15). By the

distribution laws for $\otimes$, (15) is equivalent to

$$(\forall t, r.\, \{\mathsf{emp}\}\, C(t,r)\, \{\mathsf{emp}\}) \otimes \mathit{AssocList}$$

Now we have moved the association list *AssocList* outside the triple as a framed invariant. By monotonicity of $\otimes$ we will have the required $\mathit{Mod}'(C) \otimes \mathit{AssocList}$ if we can show

$$\forall t, r.\, \{\mathsf{emp}\}\, C(t,r)\, \{\mathsf{emp}\} \quad \Rightarrow \quad \mathit{Mod}'(C)$$

This is mostly a matter of using ($\star$-FRAME) to add as invariants all the pieces of state that a module might access, such as *EvalCode* and $r \mapsto \_$. One neat feature of this proof is that one never needs to push an application of $\otimes$ through a μ operator; this is good because there is no simple distribution law relating $\otimes$ and μ.

Recall that in Fig. 3 we presented modules which make use of higher-order store in various ways: our modules exhibit dynamic discovery of available code, on-demand loading of required code, self-update and specialisation at runtime. We stress that these modules and behaviours can all be verified using our logic.

Remark 6.1. Code updates do not need to preserve behaviour.

We point out a key difference between our nested triples and store specifications as used in [23] to specify object methods stored on the heap. Using fixed store specifications that are expressed in the context like types are, if one overwrites some code (method) C with new code (method) D, D must meet the behavioural specification given in the store specification which is the one C fulfilled. As a consequence of store specifications, code updates must preserve behaviour. With nested triples this is not the case: we can overwrite code with other code having totally unrelated behaviour.

For example, in our main program (Fig. 2) we can change the line

$$\mathsf{let}\ \mathit{disposeTree} = \mathsf{new}\ 0\ \mathsf{in}\ \ldots$$

to[6]

$$\mathsf{let}\ \mathit{disposeTree} = \mathit{evalTree}\ \mathsf{in}\ \ldots \tag{16}$$

This causes the tree deletion code to reside, and recurse through the store, in the cell formerly occupied by the tree evaluation code; the new code does not satisfy the old specification $\mathit{EvalTriple}(\cdot)$. The proof that the tree disposal runs safely is essentially unchanged.

Note that the store specification used in [23] is one big invariant for the program under consideration (which is implicitly recursively defined). Using nested triples, however, one has to specify invariants explicitly and this leads to explicit recursive definitions of predicates in specifications.

[6] Technically the statement form $\mathsf{let}\ v = E\ \mathsf{in}\ C$ used in (16) above is not in the programming language, but can be treated as shorthand for (choosing x fresh)

$$\mathsf{let}\ x = \mathsf{new}\ E\ \mathsf{in}\ \mathsf{let}\ v = [x]\ \mathsf{in}\ (\mathsf{free}\ x\ ;\ C)$$

Remark 6.2. We have considered the following variation of the (Eval) rule:

$$\frac{\Gamma \vdash P \Rightarrow e \mapsto \{P\} \cdot (\vec{e})\,\{Q\} \star \mathrm{True}}{\Gamma \vdash \{P\}\,\mathsf{eval}\ [e](\vec{e})\,\{Q\}}\ \textsc{EvalPrime}$$

This rule would often be more convenient than the existing (Eval) rule. For example, in the proof of the DT code in Section 6.1, two uses of (Mu) were necessary, but with (EvalPrime) only one would be needed. Unfortunately, although (EvalPrime) appears very plausible, it is not validated by the ultrametric semantics model of [9]. The reason is that the validity of triples depends on the rank[7] of the heap, but using $\_ \star \mathrm{True}$ in the hypothesis means we cannot guarantee that $\{P\} \cdot (\vec{e})\,\{Q\}$ holds in a heap of sufficiently high rank. The rule holds in a step-indexed version of the ultrametric model as advocated in [24].

7. Conclusions and future work

We extended the separation logic of [9] for higher-order store, adding several features needed to reason about larger, more realistic programs, including parameter passing and more general recursive specification patterns. We classified and discussed several such specification patterns, corresponding to increasingly complex uses of recursion through the store. Finally we applied our specification patterns and rules to an example program that exploited many of the possibilities offered by higher-order store; thus we presented the first larger case study conducted with logical techniques based on [9].

7.1. Related work

The logic of [9] on which we build is by no means the only logic which allows reasoning about higher-order store or recursion through the store; there are several others (for instance [7, 25, 16, 26, 27, 28, 29, 30]), each with a corresponding underlying semantic model. None of these papers treat the complex patterns of recursion through the store we have considered, however.

It is interesting to ask whether the specification patterns we have identified here, or adaptations thereof, could also be used with these other logics, or whether they are somehow tied to the logic of [9]. To partially answer this, we note that the main property of the model in [9] on which our development rests is the existence of various fixpoints (see the proof of Theorem 4.1). Thus it seems possible that our specification patterns can be adapted to another logic for higher order store, provided the underlying model of that logic guarantees the existence of the appropriate fixpoints.

For example, let us consider Hoare Type Theory [31], which is the foundation of the Ynot verification system [30]. Hoare Type Theory is a dependent type

[7] The rank of a heap h is the smallest k such that $\pi_k(h) = h$ for a family of projections π_n that satisfy $\bigsqcup_n \pi_n = id_{Heap}$ [9]. The rank of a heap is used to ensure that the semantics of (nested) triples are non-expansive in the worlds.

system for higher order imperative programs, in which Hoare triples are available as types. These Hoare types function much like nested Hoare triples. In [32] it is remarked that "future work includes investigating how to model recursive types, as needed for the specification of programs that recurse through the store". The authors state that "recursive types should exist, though, since admissible pers do accommodate a wide range of recursive types". Thus it is quite plausible that our specification patterns could be carried over *mutatis mutandis* to Hoare Type Theory.

For some of the logics mentioned above, such as [7, 26], the underlying model does not ensure the existence of the required fixpoints. Of course, this does not mean that such logics are necessarily incapable of reasoning about the intricate uses of recursion through the store we have examined; it simply shows that the approach we have used here cannot be carried over straightforwardly.

7.1.1. Tool support

The complexity of the involved specifications and proofs demands tool support. We have developed a verification tool named Crowfoot [10] supporting a logic and a language very similar to that used in this paper. Crowfoot uses *symbolic execution*, as used in tools such as Smallfoot [33], jStar [34] and VeriFast [35]. Use of $\star$-FRAME by Crowfoot is automatic, whereas the use of $\otimes$-FRAME is presently guided by user annotations.

We have used Crowfoot, for example, to verify correctness of runtime module updates [18], to verify the Reflective Visitor design pattern [36], and to verify a large subset of the example presented in this paper. The online version of Crowfoot (available on our website [37]) provides the code and specifications (plus proof hints) for the main program and modules PLUS, WHILE, OSCILLATE, and LOAD_OVERWRITE.

Since Crowfoot is a proof-of-concept tool for higher-order store it does not (yet) support the division operator and thus the modules involving complex arithmetic (COPRIME and BINOM) could not be ported to Crowfoot. Moreover, Crowfoot cannot express triples for code stored in program variables, only for code stored on the heap. Therefore, the implementation of the OSCILLATE module has been slightly altered in the Crowfoot version.

7.2. Future work

The results of this paper raise some interesting new challenges. For instance, we were able to perform specialisation at runtime of the $\binom{n}{k}$ operator just by exploiting the static scoping of the programming language, but for other cases this will be insufficient. We plan to extend our language and logic with (extensional) operations for generating code at runtime.

In proving our example program, we used the ($\otimes$-FRAME) rule. However, ($\otimes$-FRAME) gives only a limited form of information hiding: for instance it does not allow us to prove a module which sets up some persistent state on its first execution, and then uses it on subsequent invocations. The recent paper [38] extends the logic of [9] by adding an *anti-frame* rule, which provides more

flexible information hiding. It has not been considered in our example here. Because most of our proof rules are justified by reducing them via syntactic means to those of [9], it is very likely that the extensions we introduced are compatible with the anti-frame rule, but this is yet to be proved formally.

Acknowledgements. This research was supported by the EPSRC grant *From Reasoning Principles for Function Pointers To Logics for Self-Configuring Programs* (EP/G003173/1).

Appendix A. Proof for the PLUS module

In this appendix we give the proof for the soundness of the PLUS module. Due to lack of horizontal space we often present triples in the form:

$$\begin{array}{c}\{\ \text{long pre-condition}\ \}\ C\ \{\ \text{long post-condition}\ \}\\ C = \\ \text{program statement}_1;\\ \text{program statement}_2;\\ \text{program statement}_3\end{array}$$

instead of the usual triple layout

$$\{\ \text{long pre-condition}\ \}\ \begin{array}{l}\text{program statement}_1;\\ \text{program statement}_2;\\ \text{program statement}_3\end{array}\ \{\ \text{long post-condition}\ \}.$$

As we remarked, we can prove $Mod'(M)$ (where M is the code of PLUS) and then use $\otimes$-FRAME to get $Mod'(M) \otimes AssocList$. So we need to prove:

$$\begin{array}{l}\forall t, r, \tau_1, \sigma_1\ .\\ \left\{\begin{array}{l}\exists a.\quad \tau_1 \subseteq \sigma_1 \wedge\\ \quad modlist \mapsto a\\ \star\ CLseg(a, \mathsf{null}, \sigma_1)\\ \star\ \mathrm{tree}_{\mathrm{fork}}(t, \tau_1)\\ \star\ EvalCode\\ \star\ LoaderCode\\ \star\ SearchModsCode\\ \star\ r \mapsto \_\end{array}\right\}\ C\ \left\{\begin{array}{l}\exists a, \sigma_2, \tau_2.\ \tau_2 \subseteq \sigma_2\ \wedge\ \sigma_1 \subseteq \sigma_2 \wedge\\ \quad modlist \mapsto a\\ \star\ CLseg(a, \mathsf{null}, \sigma_2)\\ \star\ \mathrm{tree}(t, \tau_2)\\ \star\ EvalCode\\ \star\ LoaderCode\\ \star\ SearchModsCode\\ \star\ r \mapsto \_\end{array}\right\}\\ C = \text{`}\\ \quad \lambda tree, resaddr.\\ \quad\ \mathsf{let}\ left = [tree + \mathrm{LeftO}]\ \mathsf{in}\\ \quad\ \mathsf{let}\ right = [tree + \mathrm{RightO}]\ \mathsf{in}\\ \quad\quad \mathsf{eval}\ [evalTree](left, \mathsf{res}\ leftVal)\ ;\\ \quad\quad \mathsf{eval}\ [evalTree](right, \mathsf{res}\ rightVal)\ ;\\ \quad\quad [resaddr] := leftVal + rightVal\\ \text{'}\ (t, r)\end{array}$$

By universal generalisation and the LAMBDA rule, it will suffice to prove:

$$\left\{\begin{array}{l}\exists a\ .\\ \quad \mathit{modlist} \mapsto a\\ \star\ \mathit{CLseg}(a, \mathsf{null}, \sigma_1)\\ \star\ \mathrm{tree}_{\mathrm{fork}}(\mathit{tree}, \tau_1)\\ \star\ \mathit{EvalCode}\\ \star\ \mathit{LoaderCode}\\ \star\ \mathit{SearchModsCode}\\ \star\ \mathit{resaddr} \mapsto \_\\ \wedge\ \tau_1 \subseteq \sigma_1\end{array}\right\}\ C\ \left\{\begin{array}{l}\exists a, \sigma_2, \tau_2\ .\\ \quad \mathit{modlist} \mapsto a\\ \star\ \mathit{CLseg}(a, \mathsf{null}, \sigma_2)\\ \star\ \mathrm{tree}(\mathit{tree}, \tau_2)\\ \star\ \mathit{EvalCode}\\ \star\ \mathit{LoaderCode}\\ \star\ \mathit{SearchModsCode}\\ \star\ \mathit{resaddr} \mapsto \_\\ \wedge\ \tau_2 \subseteq \sigma_2\\ \wedge\ \sigma_1 \subseteq \sigma_2\end{array}\right\}$$

$$\begin{array}{l}C =\\ \mathsf{let}\ \mathit{left} = [\mathit{tree} + \mathrm{LeftO}]\ \mathsf{in}\\ \mathsf{let}\ \mathit{right} = [\mathit{tree} + \mathrm{RightO}]\ \mathsf{in}\\ \quad \mathsf{eval}\ [\mathit{evalTree}](\mathit{left}, \mathsf{res}\ \mathit{leftVal})\ ;\\ \quad \mathsf{eval}\ [\mathit{evalTree}](\mathit{right}, \mathsf{res}\ \mathit{rightVal})\ ;\\ \quad [\mathit{resaddr}] := \mathit{leftVal} + \mathit{rightVal}\end{array}$$

From now on for ease of presentation we will use the following abbreviation:

$$\mathit{Stuff} \quad := \quad \mathit{EvalCode} \star \mathit{LoaderCode} \star \mathit{SearchModsCode} \star \mathit{resaddr} \mapsto \_$$

Stuff will be an invariant in many of our triples. Expanding the abbreviation $\mathrm{tree}_{\mathrm{fork}}$, it suffices to prove:

$$\left\{\begin{array}{l}\exists a, \mathit{cPtr}, \mathit{left}, \mathit{right}, \tau_{\mathrm{le}}, \tau_{\mathrm{ri}}\ .\\ \quad \mathit{modlist} \mapsto a\\ \star\ \mathit{CLseg}(a, \mathsf{null}, \sigma_1)\\ \star\ \mathit{tree} \mapsto \mathrm{opLbl}, \mathit{cPtr}, \mathit{left}, \mathit{right}, \_\\ \star\ \mathrm{tree}(\mathit{left}, \tau_{\mathrm{le}})\\ \star\ \mathrm{tree}(\mathit{right}, \tau_{\mathrm{ri}})\\ \star\ \mathit{Stuff}\\ \wedge\ \tau_1 \subseteq \sigma_1\\ \wedge\ \tau_1 = \{\mathit{cPtr}\} \cup \tau_{\mathrm{le}} \cup \tau_{\mathrm{ri}}\end{array}\right\}\ C\ \left\{\begin{array}{l}\exists a, \sigma_2, \tau_2\ .\\ \quad \mathit{modlist} \mapsto a\\ \star\ \mathit{CLseg}(a, \mathsf{null}, \sigma_2)\\ \star\ \mathrm{tree}(\mathit{tree}, \tau_2)\\ \star\ \mathit{Stuff}\\ \wedge\ \tau_2 \subseteq \sigma_2\\ \wedge\ \sigma_1 \subseteq \sigma_2\end{array}\right\}$$

$$\begin{array}{l}C =\\ \mathsf{let}\ \mathit{left} = [\mathit{tree} + \mathrm{LeftO}]\ \mathsf{in}\\ \mathsf{let}\ \mathit{right} = [\mathit{tree} + \mathrm{RightO}]\ \mathsf{in}\\ \quad \mathsf{eval}\ [\mathit{evalTree}](\mathit{left}, \mathsf{res}\ \mathit{leftVal});\\ \quad \mathsf{eval}\ [\mathit{evalTree}](\mathit{right}, \mathsf{res}\ \mathit{rightVal});\\ \quad [\mathit{resaddr}] := \mathit{leftVal} + \mathit{rightVal}\end{array}$$

Standard reasoning reduces this to the following:

$$\left\{\begin{array}{l}\exists a, cPtr, \tau_{\mathrm{le}}, \tau_{\mathrm{ri}}\ . \\ \quad modlist \mapsto a \\ \star\ CLseg(a, \mathsf{null}, \sigma_1) \\ \star\ tree \mapsto \mathrm{opLbl}, cPtr, left, right, \_ \\ \star\ \mathrm{tree}(left, \tau_{\mathrm{le}}) \\ \star\ \mathrm{tree}(right, \tau_{\mathrm{ri}}) \\ \star\ Stuff \\ \wedge\ \tau_1 \subseteq \sigma_1 \\ \wedge\ \tau_1 = \{cPtr\} \cup \tau_{\mathrm{le}} \cup \tau_{\mathrm{ri}}\end{array}\right\}\ C\ \left\{\begin{array}{l}\exists a, \sigma_2, \tau_2\ . \\ \quad modlist \mapsto a \\ \star\ CLseg(a, \mathsf{null}, \sigma_2) \\ \star\ \mathrm{tree}(tree, \tau_2) \\ \star\ Stuff \\ \wedge\ \tau_2 \subseteq \sigma_2 \\ \wedge\ \sigma_1 \subseteq \sigma_2\end{array}\right\}$$

$$\begin{array}{l}C = \\ \mathsf{eval}\ [evalTree](left, \mathsf{res}\ leftVal); \\ \mathsf{eval}\ [evalTree](right, \mathsf{res}\ rightVal); \\ [resaddr] := leftVal + rightVal\end{array}$$

The existential quantifiers over *left* and *right* have disappeared. Now we expand the res abbreviation, leaving:

$$\left\{\begin{array}{l}\exists a, cPtr, \tau_{\mathrm{le}}, \tau_{\mathrm{ri}}\ . \\ \quad modlist \mapsto a \\ \star\ CLseg(a, \mathsf{null}, \sigma_1) \\ \star\ tree \mapsto \mathrm{opLbl}, cPtr, left, right, \_ \\ \star\ \mathrm{tree}(left, \tau_{\mathrm{le}}) \\ \star\ \mathrm{tree}(right, \tau_{\mathrm{ri}}) \\ \star\ Stuff \\ \wedge\ \tau_1 \subseteq \sigma_1 \\ \wedge\ \tau_1 = \{cPtr\} \cup \tau_{\mathrm{le}} \cup \tau_{\mathrm{ri}}\end{array}\right\}\ C\ \left\{\begin{array}{l}\exists a, \sigma_2, \tau_2\ . \\ \quad modlist \mapsto a \\ \star\ CLseg(a, \mathsf{null}, \sigma_2) \\ \star\ \mathrm{tree}(tree, \tau_2) \\ \star\ Stuff \\ \wedge\ \tau_2 \subseteq \sigma_2 \\ \wedge\ \sigma_1 \subseteq \sigma_2\end{array}\right\}$$

$$\begin{array}{l}C = \\ \mathsf{let}\ leftValaddr = \mathsf{new}\ 0\ \mathsf{in} \\ \quad \mathsf{eval}\ [evalTree](left, leftValaddr); \\ \quad \mathsf{let}\ leftVal = [leftValaddr]\ \mathsf{in} \\ \quad \mathsf{let}\ rightValaddr = \mathsf{new}\ 0\ \mathsf{in} \\ \qquad \mathsf{eval}\ [evalTree](right, rightValaddr); \\ \qquad \mathsf{let}\ rightVal = [rightValaddr]\ \mathsf{in} \\ \qquad\quad [resaddr] := leftVal + rightVal; \\ \qquad\quad \mathsf{free}\ leftValaddr; \\ \qquad\quad \mathsf{free}\ rightValaddr\end{array}$$

Standard reasoning deals with the let leaving:

$$\left\{\begin{array}{l}\exists a, cPtr, \tau_{\mathrm{le}}, \tau_{\mathrm{ri}}\ . \\ \quad modlist \mapsto a \\ \star\ CLseg(a, \mathsf{null}, \sigma_1) \\ \star\ tree \mapsto \mathrm{opLbl}, cPtr, left, right, \_ \\ \star\ \mathrm{tree}(left, \tau_{\mathrm{le}}) \\ \star\ \mathrm{tree}(right, \tau_{\mathrm{ri}}) \\ \star\ Stuff \\ \star\ leftValaddr \mapsto \_ \\ \wedge\ \tau_1 \subseteq \sigma_1 \\ \wedge\ \tau_1 = \{cPtr\} \cup \tau_{\mathrm{le}} \cup \tau_{\mathrm{ri}}\end{array}\right\}\ C\ \left\{\begin{array}{l}\exists a, \sigma_2, \tau_2\ . \\ \quad modlist \mapsto a \\ \star\ CLseg(a, \mathsf{null}, \sigma_2) \\ \star\ \mathrm{tree}(tree, \tau_2) \\ \star\ Stuff \\ \wedge\ \tau_2 \subseteq \sigma_2 \\ \wedge\ \sigma_1 \subseteq \sigma_2\end{array}\right\}$$

$$\begin{array}{l} C = \\ \mathsf{eval}\ [evalTree](left, leftValaddr); \\ \mathsf{let}\ leftVal = [leftValaddr]\ \mathsf{in} \\ \mathsf{let}\ rightValaddr = \mathsf{new}\ 0\ \mathsf{in} \\ \quad \mathsf{eval}\ [evalTree](right, rightValaddr); \\ \quad \mathsf{let}\ rightVal = [rightValaddr]\ \mathsf{in} \\ \qquad [resaddr] := leftVal + rightVal; \\ \qquad \mathsf{free}\ leftValaddr;\ \mathsf{free}\ rightValaddr \end{array}$$

Let us now assume that we have the following triple; we will come back and prove it later.

$$\left\{\begin{array}{l}\exists a, cPtr, \tau_{\mathrm{le}}, \tau_{\mathrm{ri}}\ . \\ \quad modlist \mapsto a \\ \star\ CLseg(a, \mathsf{null}, \sigma_1) \\ \star\ tree \mapsto \mathrm{opLbl}, cPtr, left, right, \_ \\ \star\ \mathrm{tree}(left, \tau_{\mathrm{le}}) \\ \star\ \mathrm{tree}(right, \tau_{\mathrm{ri}}) \\ \star\ Stuff \\ \star\ leftValaddr \mapsto \_ \\ \wedge\ \tau_1 \subseteq \sigma_1 \\ \wedge\ \tau_1 = \{cPtr\} \cup \tau_{\mathrm{le}} \cup \tau_{\mathrm{ri}}\end{array}\right\}\ C_{\mathrm{le}}\ \left\{\begin{array}{l}\exists a, \sigma_2, \tau_2, cPtr, \tau_{\mathrm{le}}, \tau_{\mathrm{ri}}\ . \\ \quad modlist \mapsto a \\ \star\ CLseg(a, \mathsf{null}, \sigma_2) \\ \star\ tree \mapsto \mathrm{opLbl}, cPtr, left, right, \_ \\ \star\ \mathrm{tree}(left, \tau_{\mathrm{le}}) \\ \star\ \mathrm{tree}(right, \tau_{\mathrm{ri}}) \\ \star\ Stuff \\ \star\ leftValaddr \mapsto \_ \\ \wedge\ \sigma_1 \subseteq \sigma_2 \\ \wedge\ \tau_2 \subseteq \sigma_2 \\ \wedge\ \tau_2 = \{cPtr\} \cup \tau_{\mathrm{le}} \cup \tau_{\mathrm{ri}}\end{array}\right\} \tag{A.1}$$

$$C_{\mathrm{le}} = \mathsf{eval}\ [evalTree](left, leftValaddr)$$

Using (A.1) and sequential composition, our proof obligation reduces to

$$\left\{\begin{array}{l}\exists a, \sigma_2, \tau_2, cPtr, \tau_{\mathrm{le}}, \tau_{\mathrm{ri}}\ . \\ \quad modlist \mapsto a \\ \star\ CLseg(a, \mathsf{null}, \sigma_2) \\ \star\ tree \mapsto \mathrm{opLbl}, cPtr, left, right, \_ \\ \star\ \mathrm{tree}(left, \tau_{\mathrm{le}}) \\ \star\ \mathrm{tree}(right, \tau_{\mathrm{ri}}) \\ \star\ Stuff \\ \star\ leftValaddr \mapsto \_ \\ \wedge\ \sigma_1 \subseteq \sigma_2 \\ \wedge\ \tau_2 \subseteq \sigma_2 \\ \wedge\ \tau_2 = \{cPtr\} \cup \tau_{\mathrm{le}} \cup \tau_{\mathrm{ri}}\end{array}\right\}\ C\ \left\{\begin{array}{l}\exists a, \sigma_2, \tau_2\ . \\ \quad modlist \mapsto a \\ \star\ CLseg(a, \mathsf{null}, \sigma_2) \\ \star\ \mathrm{tree}(tree, \tau_2) \\ \star\ Stuff \\ \wedge\ \tau_2 \subseteq \sigma_2 \\ \wedge\ \sigma_1 \subseteq \sigma_2\end{array}\right\}$$

$$\begin{array}{l} C = \\ \mathsf{let}\ leftVal = [leftValaddr]\ \mathsf{in} \\ \mathsf{let}\ rightValaddr = \mathsf{new}\ 0\ \mathsf{in} \\ \quad \mathsf{eval}\ [evalTree] \\ \quad\ (right, rightValaddr); \\ \quad \mathsf{let}\ rightVal = [rightValaddr]\ \mathsf{in} \\ \quad\ [resaddr] := leftVal + rightVal; \\ \quad\ \mathsf{free}\ leftValaddr; \\ \quad\ \mathsf{free}\ rightValaddr \end{array}$$

Standard reasoning reduces this further; it remains to show

$$\left\{\begin{array}{l}\exists a, \sigma_2, \tau_2, cPtr, \tau_{\mathrm{le}}, \tau_{\mathrm{ri}}\ . \\ \quad modlist \mapsto a \\ \star\ CLseg(a, \mathsf{null}, \sigma_2) \\ \star\ tree \mapsto \mathrm{opLbl}, cPtr, left, right, \_ \\ \star\ \mathrm{tree}(left, \tau_{\mathrm{le}}) \\ \star\ \mathrm{tree}(right, \tau_{\mathrm{ri}}) \\ \star\ Stuff \\ \star\ leftValaddr \mapsto \_ \\ \star\ rightValaddr \mapsto \_ \\ \wedge\ \sigma_1 \subseteq \sigma_2 \\ \wedge\ \tau_2 \subseteq \sigma_2 \\ \wedge\ \tau_2 = \{cPtr\} \cup \tau_{\mathrm{le}} \cup \tau_{\mathrm{ri}}\end{array}\right\}\ C\ \left\{\begin{array}{l}\exists a, \sigma_2, \tau_2\ . \\ \quad modlist \mapsto a \\ \star\ CLseg(a, \mathsf{null}, \sigma_2) \\ \star\ \mathrm{tree}(tree, \tau_2) \\ \star\ Stuff \\ \wedge\ \tau_2 \subseteq \sigma_2 \\ \wedge\ \sigma_1 \subseteq \sigma_2\end{array}\right\}$$

$$\begin{array}{l} C = \\ \quad \mathsf{eval}\ [evalTree] \\ \quad\ (right, rightValaddr); \\ \quad \mathsf{let}\ rightVal = [rightValaddr]\ \mathsf{in} \\ \quad\ [resaddr] := leftVal + rightVal; \\ \quad\ \mathsf{free}\ leftValaddr\ ; \\ \quad\ \mathsf{free}\ rightValaddr \end{array}$$

Again we will assume a triple for the eval, deferring its proof until later. We assume:

$$\left\{\begin{array}{l}\exists a, \sigma_2, \tau_2, cPtr, \tau_{\mathrm{le}}, \tau_{\mathrm{ri}}\ . \\ \quad modlist \mapsto a \\ \star\ CLseg(a, \mathsf{null}, \sigma_2) \\ \star\ tree \mapsto \mathrm{opLbl}, cPtr, left, right, \_ \\ \star\ \mathrm{tree}(left, \tau_{\mathrm{le}}) \\ \star\ \mathrm{tree}(right, \tau_{\mathrm{ri}}) \\ \star\ Stuff \\ \star\ leftValaddr \mapsto \_ \\ \star\ rightValaddr \mapsto \_ \\ \wedge\ \sigma_1 \subseteq \sigma_2 \\ \wedge\ \tau_2 \subseteq \sigma_2 \\ \wedge\ \tau_2 = \{cPtr\} \cup \tau_{\mathrm{le}} \cup \tau_{\mathrm{ri}}\end{array}\right\}\ C_{\mathrm{ri}}\ \left\{\begin{array}{l}\exists a, \sigma_2, \tau_2, cPtr, \tau_{\mathrm{le}}, \tau_{\mathrm{ri}}\ . \\ \quad modlist \mapsto a \\ \star\ CLseg(a, \mathsf{null}, \sigma_2) \\ \star\ tree \mapsto \mathrm{opLbl}, cPtr, left, right, \_ \\ \star\ \mathrm{tree}(left, \tau_{\mathrm{le}}) \\ \star\ \mathrm{tree}(right, \tau_{\mathrm{ri}}) \\ \star\ Stuff \\ \star\ leftValaddr \mapsto \_ \\ \star\ rightValaddr \mapsto \_ \\ \wedge\ \sigma_1 \subseteq \sigma_2 \\ \wedge\ \tau_2 \subseteq \sigma_2 \\ \wedge\ \tau_2 = \{cPtr\} \cup \tau_{\mathrm{le}} \cup \tau_{\mathrm{ri}}\end{array}\right\} \tag{A.2}$$

$$C_{\mathrm{ri}} = \mathsf{eval}\ [evalTree](right, rightValaddr)$$

Then by sequential composition it remains to prove:

$$\left\{\begin{array}{l}\exists a, \sigma_2, \tau_2, cPtr, \tau_{\mathrm{le}}, \tau_{\mathrm{ri}}\ . \\ \quad modlist \mapsto a \\ \star\ CLseg(a, \mathsf{null}, \sigma_2) \\ \star\ tree \mapsto \mathrm{opLbl}, cPtr, left, right, \_ \\ \star\ \mathrm{tree}(left, \tau_{\mathrm{le}}) \\ \star\ \mathrm{tree}(right, \tau_{\mathrm{ri}}) \\ \star\ Stuff \\ \star\ leftValaddr \mapsto \_ \\ \star\ rightValaddr \mapsto \_ \\ \wedge\ \sigma_1 \subseteq \sigma_2 \\ \wedge\ \tau_2 \subseteq \sigma_2 \\ \wedge\ \tau_2 = \{cPtr\} \cup \tau_{\mathrm{le}} \cup \tau_{\mathrm{ri}}\end{array}\right\}\ C\ \left\{\begin{array}{l}\exists a, \sigma_2, \tau_2\ . \\ \quad modlist \mapsto a \\ \star\ CLseg(a, \mathsf{null}, \sigma_2) \\ \star\ \mathrm{tree}(tree, \tau_2) \\ \star\ Stuff \\ \wedge\ \tau_2 \subseteq \sigma_2 \\ \wedge\ \sigma_1 \subseteq \sigma_2\end{array}\right\}$$

$$\begin{array}{l} C = \\ \quad \mathsf{let}\ rightVal = [rightValaddr]\ \mathsf{in} \\ \quad\quad [resaddr] := leftVal + rightVal; \\ \quad\quad \mathsf{free}\ leftValaddr; \\ \quad\quad \mathsf{free}\ rightValaddr \end{array}$$

By standard reasoning this would follow from

$$\left\{\begin{array}{l}\exists a, \sigma_2, \tau_2, \mathit{cPtr}, \tau_{\mathrm{le}}, \tau_{\mathrm{ri}}\ .\\ \quad \mathit{modlist} \mapsto a\\ \star\ \mathit{CLseg}(a, \mathsf{null}, \sigma_2)\\ \star\ \mathit{tree} \mapsto \mathrm{opLbl}, \mathit{cPtr}, \mathit{left}, \mathit{right}, \_\\ \star\ \mathrm{tree}(\mathit{left}, \tau_{\mathrm{le}})\\ \star\ \mathrm{tree}(\mathit{right}, \tau_{\mathrm{ri}})\\ \star\ \mathit{Stuff}\\ \wedge\ \sigma_1 \subseteq \sigma_2\\ \wedge\ \tau_2 \subseteq \sigma_2\\ \wedge\ \tau_2 = \{\mathit{cPtr}\} \cup \tau_{\mathrm{le}} \cup \tau_{\mathrm{ri}}\end{array}\right\}\ \mathsf{skip}\ \left\{\begin{array}{l}\exists a, \sigma_2, \tau_2\ .\\ \quad \mathit{modlist} \mapsto a\\ \star\ \mathit{CLseg}(a, \mathsf{null}, \sigma_2)\\ \star\ \mathrm{tree}(\mathit{tree}, \tau_2)\\ \star\ \mathit{Stuff}\\ \wedge\ \tau_2 \subseteq \sigma_2\\ \wedge\ \sigma_1 \subseteq \sigma_2\end{array}\right\}$$

We now use the consequence rule to weaken the precondition, introducing existential quantifiers over *left* and *right* again; it is enough to prove

$$\left\{\begin{array}{l}\exists a, \sigma_2, \tau_2, \mathit{cPtr}, \mathit{left}, \mathit{right}, \tau_{\mathrm{le}}, \tau_{\mathrm{ri}}\ .\\ \quad \mathit{modlist} \mapsto a\\ \star\ \mathit{CLseg}(a, \mathsf{null}, \sigma_2)\\ \star\ \mathit{tree} \mapsto \mathrm{opLbl}, \mathit{cPtr}, \mathit{left}, \mathit{right}, \_\\ \star\ \mathrm{tree}(\mathit{left}, \tau_{\mathrm{le}})\\ \star\ \mathrm{tree}(\mathit{right}, \tau_{\mathrm{ri}})\\ \star\ \mathit{Stuff}\\ \wedge\ \sigma_1 \subseteq \sigma_2\\ \wedge\ \tau_2 \subseteq \sigma_2\\ \wedge\ \tau_2 = \{\mathit{cPtr}\} \cup \tau_{\mathrm{le}} \cup \tau_{\mathrm{ri}}\end{array}\right\}\ \mathsf{skip}\ \left\{\begin{array}{l}\exists a, \sigma_2, \tau_2\ .\\ \quad \mathit{modlist} \mapsto a\\ \star\ \mathit{CLseg}(a, \mathsf{null}, \sigma_2)\\ \star\ \mathrm{tree}(\mathit{tree}, \tau_2)\\ \star\ \mathit{Stuff}\\ \wedge\ \tau_2 \subseteq \sigma_2\\ \wedge\ \sigma_1 \subseteq \sigma_2\end{array}\right\}$$

Restoring the abbreviation $\mathrm{tree}_{\mathrm{fork}}$, we are left with

$$\left\{\begin{array}{l}\exists a, \sigma_2, \tau_2\ .\\ \quad \mathit{modlist} \mapsto a\\ \star\ \mathit{CLseg}(a, \mathsf{null}, \sigma_2)\\ \star\ \mathrm{tree}_{\mathrm{fork}}(\mathit{tree}, \tau_2)\\ \star\ \mathit{Stuff}\\ \wedge\ \sigma_1 \subseteq \sigma_2\\ \wedge\ \tau_2 \subseteq \sigma_2\end{array}\right\}\ \mathsf{skip}\ \left\{\begin{array}{l}\exists a, \sigma_2, \tau_2\ .\\ \quad \mathit{modlist} \mapsto a\\ \star\ \mathit{CLseg}(a, \mathsf{null}, \sigma_2)\\ \star\ \mathrm{tree}(\mathit{tree}, \tau_2)\\ \star\ \mathit{Stuff}\\ \wedge\ \tau_2 \subseteq \sigma_2\\ \wedge\ \sigma_1 \subseteq \sigma_2\end{array}\right\}$$

This holds because from the definition of tree we have $\mathrm{tree}_{\mathrm{fork}}(t, \tau) \Rightarrow \mathrm{tree}(t, \tau)$.

Let us now prove (A.1). Formally, we first use the (Mu) rule in the precondition to unfold the *EvalCode* predicate (which was hidden in the *Stuff* abbreviation), leaving

$$\left\{\begin{array}{l}\exists a, cPtr, \tau_{\mathrm{le}}, \tau_{\mathrm{ri}}\ . \\ \quad modlist \mapsto a \\ \star\ CLseg(a, \mathsf{null}, \sigma_1) \\ \star\ tree \mapsto \mathrm{opLbl}, cPtr, left, right, \_ \\ \star\ \mathrm{tree}(left, \tau_{\mathrm{le}}) \\ \star\ \mathrm{tree}(right, \tau_{\mathrm{ri}}) \\ \star\ evalTree \mapsto EvalTriple(\cdot) \\ \star\ LoaderCode \\ \star\ SearchModsCode \\ \star\ resaddr \mapsto \_ \\ \star\ leftValaddr \mapsto \_ \\ \wedge\ \tau_1 \subseteq \sigma_1 \\ \wedge\ \tau_1 = \{cPtr\} \cup \tau_{\mathrm{le}} \cup \tau_{\mathrm{ri}}\end{array}\right\}\ C\ \left\{\begin{array}{l}\exists a, \sigma_2, \tau_2, cPtr, \tau_{\mathrm{le}}, \tau_{\mathrm{ri}}\ . \\ \quad modlist \mapsto a \\ \star\ CLseg(a, \mathsf{null}, \sigma_2) \\ \star\ tree \mapsto \mathrm{opLbl}, cPtr, left, right, \_ \\ \star\ \mathrm{tree}(left, \tau_{\mathrm{le}}) \\ \star\ \mathrm{tree}(right, \tau_{\mathrm{ri}}) \\ \star\ Stuff \\ \star\ leftValaddr \mapsto \_ \\ \wedge\ \sigma_1 \subseteq \sigma_2 \\ \wedge\ \tau_2 \subseteq \sigma_2 \\ \wedge\ \tau_2 = \{cPtr\} \cup \tau_{\mathrm{le}} \cup \tau_{\mathrm{ri}}\end{array}\right\}$$

$$C = \mathsf{eval}\ [evalTree](left, leftValaddr)$$

We can apply the (Eval) rule; we'll be done if we can show the following implication:

$$EvalTriple(k) \Rightarrow \tag{A.3}$$

$$\left\{\begin{array}{l}\exists a, cPtr, \tau_{\mathrm{le}}, \tau_{\mathrm{ri}}\ . \\ \quad modlist \mapsto a \\ \star\ CLseg(a, \mathsf{null}, \sigma_1) \\ \star\ tree \mapsto \mathrm{opLbl}, cPtr, left, right, \_ \\ \star\ \mathrm{tree}(left, \tau_{\mathrm{le}}) \\ \star\ \mathrm{tree}(right, \tau_{\mathrm{ri}}) \\ \star\ evalTree \mapsto EvalTriple(\cdot) \\ \star\ LoaderCode \\ \star\ SearchModsCode \\ \star\ resaddr \mapsto \_ \\ \star\ leftValaddr \mapsto \_ \\ \wedge\ \tau_1 \subseteq \sigma_1 \\ \wedge\ \tau_1 = \{cPtr\} \cup \tau_{\mathrm{le}} \cup \tau_{\mathrm{ri}}\end{array}\right\}\ C\ \left\{\begin{array}{l}\exists a, \sigma_2, \tau_2, cPtr, \tau_{\mathrm{le}}, \tau_{\mathrm{ri}}\ . \\ \quad modlist \mapsto a \\ \star\ CLseg(a, \mathsf{null}, \sigma_2) \\ \star\ tree \mapsto \mathrm{opLbl}, cPtr, left, right, \_ \\ \star\ \mathrm{tree}(left, \tau_{\mathrm{le}}) \\ \star\ \mathrm{tree}(right, \tau_{\mathrm{ri}}) \\ \star\ EvalCode \\ \star\ LoaderCode \\ \star\ SearchModsCode \\ \star\ resaddr \mapsto \_ \\ \star\ leftValaddr \mapsto \_ \\ \wedge\ \sigma_1 \subseteq \sigma_2 \\ \wedge\ \tau_2 \subseteq \sigma_2 \\ \wedge\ \tau_2 = \{cPtr\} \cup \tau_{\mathrm{le}} \cup \tau_{\mathrm{ri}}\end{array}\right\}$$

$$C = k(left, leftValaddr)$$

So we'll start with the LHS and derive the RHS. The LHS is:

$$\forall t, r, \tau_1, \sigma_1\ .\quad \left\{\begin{array}{l}\exists a.\quad \tau_1 \subseteq \sigma_1 \wedge \\ \quad modlist \mapsto a \\ \star\ CLseg(a, \mathsf{null}, \sigma_1) \\ \star\ \mathrm{tree}(t, \tau_1) \\ \star\ EvalCode \\ \star\ LoaderCode \\ \star\ SearchModsCode \\ \star\ r \mapsto \_\end{array}\right\}\ k(t, r)\ \left\{\begin{array}{l}\exists a, \sigma_2, \tau_2.\quad \tau_2 \subseteq \sigma_2 \wedge \sigma_1 \subseteq \sigma_2 \wedge \\ \quad modlist \mapsto a \\ \star\ CLseg(a, \mathsf{null}, \sigma_2) \\ \star\ \mathrm{tree}(t, \tau_2) \\ \star\ EvalCode \\ \star\ LoaderCode \\ \star\ SearchModsCode \\ \star\ r \mapsto \_\end{array}\right\}$$

Instantiating the quantified variables t, r, σ_1 respectively with *left*, *leftValaddr*, σ_1, and renaming τ_1 to τ_{left}, gives

$$\forall \tau_{\text{left}} \,.\, \left\{ \begin{array}{l} \exists a. \\ \quad \mathit{modlist} \mapsto a \\ \star\ \mathit{CLseg}(a, \mathsf{null}, \sigma_1) \\ \star\ \text{tree}(\mathit{left}, \tau_{\text{left}}) \\ \star\ \mathit{EvalCode} \\ \star\ \mathit{LoaderCode} \\ \star\ \mathit{SearchModsCode} \\ \star\ \mathit{leftValaddr} \mapsto \_ \\ \wedge\ \tau_{\text{left}} \subseteq \sigma_1 \end{array} \right\} \; k(\mathit{left}, \mathit{leftValaddr}) \; \left\{ \begin{array}{l} \exists a, \sigma_2, \tau_2. \\ \quad \mathit{modlist} \mapsto a \\ \star\ \mathit{CLseg}(a, \mathsf{null}, \sigma_2) \\ \star\ \text{tree}(\mathit{left}, \tau_2) \\ \star\ \mathit{EvalCode} \\ \star\ \mathit{LoaderCode} \\ \star\ \mathit{SearchModsCode} \\ \star\ \mathit{leftValaddr} \mapsto \_ \\ \wedge\ \tau_2 \subseteq \sigma_2 \\ \wedge\ \sigma_1 \subseteq \sigma_2 \end{array} \right\}$$

The variables *cPtr* and τ_{right} do not appear in this triple, so we can add a universal quantifier over them, giving:

$$\forall \mathit{cPtr}, \tau_{\text{left}}, \tau_{\text{right}}.$$
$$\left\{ \begin{array}{l} \exists a. \\ \quad \mathit{modlist} \mapsto a \\ \star\ \mathit{CLseg}(a, \mathsf{null}, \sigma_1) \\ \star\ \text{tree}(\mathit{left}, \tau_{\text{left}}) \\ \star\ \mathit{EvalCode} \\ \star\ \mathit{LoaderCode} \\ \star\ \mathit{SearchModsCode} \\ \star\ \mathit{leftValaddr} \mapsto \_ \\ \wedge\ \tau_{\text{left}} \subseteq \sigma_1 \end{array} \right\} \; k(\mathit{left}, \mathit{leftValaddr}) \; \left\{ \begin{array}{l} \exists a, \sigma_2, \tau_2. \\ \quad \mathit{modlist} \mapsto a \\ \star\ \mathit{CLseg}(a, \mathsf{null}, \sigma_2) \\ \star\ \text{tree}(\mathit{left}, \tau_2) \\ \star\ \mathit{EvalCode} \\ \star\ \mathit{LoaderCode} \\ \star\ \mathit{SearchModsCode} \\ \star\ \mathit{leftValaddr} \mapsto \_ \\ \wedge\ \tau_2 \subseteq \sigma_2 \\ \wedge\ \sigma_1 \subseteq \sigma_2 \end{array} \right\}$$

Next we use the $\star$-FRAME axiom to add the following things

$$\begin{array}{l} \quad \mathit{tree} \mapsto \text{opLbl}, \mathit{cPtr}, \mathit{left}, \mathit{right}, \_ \\ \star\ \text{tree}(\mathit{right}, \tau_{\text{right}}) \\ \star\ \mathit{resaddr} \mapsto \_ \\ \wedge\ \tau_1 \subseteq \sigma_1 \\ \wedge\ \tau_1 = \{\mathit{cPtr}\} \cup \tau_{\text{left}} \cup \tau_{\text{right}} \end{array}$$

to the pre- and post-conditions, resulting in the following triple.

$\forall cPtr, \tau_{\text{left}}, \tau_{\text{right}}\ .$

$$\left\{\begin{array}{l}\exists a.\\ \quad \mathit{modlist} \mapsto a\\ \star\ \mathit{CLseg}(a, \mathsf{null}, \sigma_1)\\ \star\ \mathit{tree} \mapsto \text{opLbl}, \mathit{cPtr}, \mathit{left}, \mathit{right}, \_\\ \star\ \text{tree}(\mathit{left}, \tau_{\text{left}})\\ \star\ \text{tree}(\mathit{right}, \tau_{\text{right}})\\ \star\ \mathit{Stuff}\\ \star\ \mathit{leftValaddr} \mapsto \_\\ \wedge\ \tau_{\text{left}} \subseteq \sigma_1\\ \wedge\ \tau_1 \subseteq \sigma_1\\ \wedge\ \tau_1 = \{\mathit{cPtr}\} \cup \tau_{\text{left}} \cup \tau_{\text{right}}\end{array}\right\}\ C\ \left\{\begin{array}{l}\exists a, \sigma_2, \tau_2.\\ \quad \mathit{modlist} \mapsto a\\ \star\ \mathit{CLseg}(a, \mathsf{null}, \sigma_2)\\ \star\ \mathit{tree} \mapsto \text{opLbl}, \mathit{cPtr}, \mathit{left}, \mathit{right}, \_\\ \star\ \text{tree}(\mathit{left}, \tau_2)\\ \star\ \text{tree}(\mathit{right}, \tau_{\text{right}})\\ \star\ \mathit{Stuff}\\ \star\ \mathit{leftValaddr} \mapsto \_\\ \wedge\ \tau_2 \subseteq \sigma_2\\ \wedge\ \sigma_1 \subseteq \sigma_2\\ \wedge\ \tau_1 \subseteq \sigma_1\\ \wedge\ \tau_1 = \{\mathit{cPtr}\} \cup \tau_{\text{left}} \cup \tau_{\text{right}}\end{array}\right\}$$

$$C = k(\mathit{left}, \mathit{leftValaddr})$$

We can drop $\tau_{\text{left}} \subseteq \sigma_1$ from the precondition because this is already implied by the other two pure constraints. We then apply (ForallExists) to all three universally quantified variables to get the following triple:

$$\left\{\begin{array}{l}\exists a, \mathit{cPtr}, \tau_{\text{left}}, \tau_{\text{right}}.\\ \quad \mathit{modlist} \mapsto a\\ \star\ \mathit{CLseg}(a, \mathsf{null}, \sigma_1)\\ \star\ \mathit{tree} \mapsto \text{opLbl}, \mathit{cPtr}, \mathit{left}, \mathit{right}, \_\\ \star\ \text{tree}(\mathit{left}, \tau_{\text{left}})\\ \star\ \text{tree}(\mathit{right}, \tau_{\text{right}})\\ \star\ \mathit{Stuff}\\ \star\ \mathit{leftValaddr} \mapsto \_\\ \wedge\ \tau_1 \subseteq \sigma_1\\ \wedge\ \tau_1 = \{\mathit{cPtr}\} \cup \tau_{\text{left}} \cup \tau_{\text{right}}\end{array}\right\}\ C\ \left\{\begin{array}{l}\exists a, \sigma_2, \tau_2, \mathit{cPtr}, \tau_{\text{left}}, \tau_{\text{right}}.\\ \quad \mathit{modlist} \mapsto a\\ \star\ \mathit{CLseg}(a, \mathsf{null}, \sigma_2)\\ \star\ \mathit{tree} \mapsto \text{opLbl}, \mathit{cPtr}, \mathit{left}, \mathit{right}, \_\\ \star\ \text{tree}(\mathit{left}, \tau_2)\\ \star\ \text{tree}(\mathit{right}, \tau_{\text{right}})\\ \star\ \mathit{Stuff}\\ \star\ \mathit{leftValaddr} \mapsto \_\\ \wedge\ \tau_2 \subseteq \sigma_2\\ \wedge\ \sigma_1 \subseteq \sigma_2\\ \wedge\ \tau_1 \subseteq \sigma_1\\ \wedge\ \tau_1 = \{\mathit{cPtr}\} \cup \tau_{\text{left}} \cup \tau_{\text{right}}\end{array}\right\}$$

In the postcondition we rename τ_{left} to τ'.

$$\left\{\begin{array}{l}\exists a, \mathit{cPtr}, \tau_{\text{left}}, \tau_{\text{right}}.\\ \quad \mathit{modlist} \mapsto a\\ \star\ \mathit{CLseg}(a, \mathsf{null}, \sigma_1)\\ \star\ \mathit{tree} \mapsto \text{opLbl}, \mathit{cPtr}, \mathit{left}, \mathit{right}, \_\\ \star\ \text{tree}(\mathit{left}, \tau_{\text{left}})\\ \star\ \text{tree}(\mathit{right}, \tau_{\text{right}})\\ \star\ \mathit{Stuff}\\ \star\ \mathit{leftValaddr} \mapsto \_\\ \wedge\ \tau_1 \subseteq \sigma_1\\ \wedge\ \tau_1 = \{\mathit{cPtr}\} \cup \tau_{\text{left}} \cup \tau_{\text{right}}\end{array}\right\}\ C\ \left\{\begin{array}{l}\exists a, \sigma_2, \tau_2, \mathit{cPtr}, \tau', \tau_{\text{right}}.\\ \quad \mathit{modlist} \mapsto a\\ \star\ \mathit{CLseg}(a, \mathsf{null}, \sigma_2)\\ \star\ \mathit{tree} \mapsto \text{opLbl}, \mathit{cPtr}, \mathit{left}, \mathit{right}, \_\\ \star\ \text{tree}(\mathit{left}, \tau_2)\\ \star\ \text{tree}(\mathit{right}, \tau_{\text{right}})\\ \star\ \mathit{Stuff}\\ \star\ \mathit{leftValaddr} \mapsto \_\\ \wedge\ \tau_2 \subseteq \sigma_2\\ \wedge\ \sigma_1 \subseteq \sigma_2\\ \wedge\ \tau_1 \subseteq \sigma_1\\ \wedge\ \tau_1 = \{\mathit{cPtr}\} \cup \tau' \cup \tau_{\text{right}}\end{array}\right\}$$

Then we rename τ_2 in the postcondition to τ_{left}.

$$\left\{\begin{array}{l}\exists a, cPtr, \tau_{\text{left}}, \tau_{\text{right}}.\\ \quad \mathit{modlist} \mapsto a\\ \star\ \mathit{CLseg}(a, \mathsf{null}, \sigma_1)\\ \star\ \mathit{tree} \mapsto \text{opLbl}, cPtr, \mathit{left}, \mathit{right}, \_\\ \star\ \text{tree}(\mathit{left}, \tau_{\text{left}})\\ \star\ \text{tree}(\mathit{right}, \tau_{\text{right}})\\ \star\ \mathit{Stuff}\\ \star\ \mathit{leftValaddr} \mapsto \_\\ \wedge\ \tau_1 \subseteq \sigma_1\\ \wedge\ \tau_1 = \{cPtr\} \cup \tau_{\text{left}} \cup \tau_{\text{right}}\end{array}\right\}\ C\ \left\{\begin{array}{l}\exists a, \sigma_2, cPtr, \tau', \tau_{\text{left}}, \tau_{\text{right}}.\\ \quad \mathit{modlist} \mapsto a\\ \star\ \mathit{CLseg}(a, \mathsf{null}, \sigma_2)\\ \star\ \mathit{tree} \mapsto \text{opLbl}, cPtr, \mathit{left}, \mathit{right}, \_\\ \star\ \text{tree}(\mathit{left}, \tau_{\text{left}})\\ \star\ \text{tree}(\mathit{right}, \tau_{\text{right}})\\ \star\ \mathit{Stuff}\\ \star\ \mathit{leftValaddr} \mapsto \_\\ \wedge\ \tau_{\text{left}} \subseteq \sigma_2\\ \wedge\ \sigma_1 \subseteq \sigma_2\\ \wedge\ \tau_1 \subseteq \sigma_1\\ \wedge\ \tau_1 = \{cPtr\} \cup \tau' \cup \tau_{\text{right}}\end{array}\right\}$$

Then in the postcondition we introduce an existentially quantified τ_2, with value exactly $\{cPtr\} \cup \tau_{\text{left}} \cup \tau_{\text{right}}$.

$$\left\{\begin{array}{l}\exists a, cPtr, \tau_{\text{left}}, \tau_{\text{right}}.\\ \quad \mathit{modlist} \mapsto a\\ \star\ \mathit{CLseg}(a, \mathsf{null}, \sigma_1)\\ \star\ \mathit{tree} \mapsto \text{opLbl}, cPtr, \mathit{left}, \mathit{right}, \_\\ \star\ \text{tree}(\mathit{left}, \tau_{\text{left}})\\ \star\ \text{tree}(\mathit{right}, \tau_{\text{right}})\\ \star\ \mathit{Stuff}\\ \star\ \mathit{leftValaddr} \mapsto \_\\ \wedge\ \tau_1 \subseteq \sigma_1\\ \wedge\ \tau_1 = \{cPtr\} \cup \tau_{\text{left}} \cup \tau_{\text{right}}\end{array}\right\}\ C\ \left\{\begin{array}{l}\exists a, \sigma_2, \tau_2, cPtr, \tau', \tau_{\text{left}}, \tau_{\text{right}}.\\ \quad \mathit{modlist} \mapsto a\\ \star\ \mathit{CLseg}(a, \mathsf{null}, \sigma_2)\\ \star\ \mathit{tree} \mapsto \text{opLbl}, cPtr, \mathit{left}, \mathit{right}, \_\\ \star\ \text{tree}(\mathit{left}, \tau_{\text{left}})\\ \star\ \text{tree}(\mathit{right}, \tau_{\text{right}})\\ \star\ \mathit{Stuff}\\ \star\ \mathit{leftValaddr} \mapsto \_\\ \wedge\ \tau_{\text{left}} \subseteq \sigma_2\\ \wedge\ \sigma_1 \subseteq \sigma_2\\ \wedge\ \tau_1 \subseteq \sigma_1\\ \wedge\ \tau_1 = \{cPtr\} \cup \tau' \cup \tau_{\text{right}}\\ \wedge\ \tau_2 = \{cPtr\} \cup \tau_{\text{left}} \cup \tau_{\text{right}}\end{array}\right\}$$

Now we can add $\tau_2 \subseteq \sigma_2$ to the postcondition, because it is already implied by the other pure constraints.

$$\left\{\begin{array}{l}\exists a, cPtr, \tau_{\text{left}}, \tau_{\text{right}}.\\ \quad \mathit{modlist} \mapsto a\\ \star\ \mathit{CLseg}(a, \mathsf{null}, \sigma_1)\\ \star\ \mathit{tree} \mapsto \text{opLbl}, cPtr, \mathit{left}, \mathit{right}, \_\\ \star\ \text{tree}(\mathit{left}, \tau_{\text{left}})\\ \star\ \text{tree}(\mathit{right}, \tau_{\text{right}})\\ \star\ \mathit{Stuff}\\ \star\ \mathit{leftValaddr} \mapsto \_\\ \wedge\ \tau_1 \subseteq \sigma_1\\ \wedge\ \tau_1 = \{cPtr\} \cup \tau_{\text{left}} \cup \tau_{\text{right}}\end{array}\right\}\ C\ \left\{\begin{array}{l}\exists a, \sigma_2, \tau_2, cPtr, \tau', \tau_{\text{left}}, \tau_{\text{right}}.\\ \quad \mathit{modlist} \mapsto a\\ \star\ \mathit{CLseg}(a, \mathsf{null}, \sigma_2)\\ \star\ \mathit{tree} \mapsto \text{opLbl}, cPtr, \mathit{left}, \mathit{right}, \_\\ \star\ \text{tree}(\mathit{left}, \tau_{\text{left}})\\ \star\ \text{tree}(\mathit{right}, \tau_{\text{right}})\\ \star\ \mathit{Stuff}\\ \star\ \mathit{leftValaddr} \mapsto \_\\ \wedge\ \tau_{\text{left}} \subseteq \sigma_2\\ \wedge\ \sigma_1 \subseteq \sigma_2\\ \wedge\ \tau_1 \subseteq \sigma_1\\ \wedge\ \tau_1 = \{cPtr\} \cup \tau' \cup \tau_{\text{right}}\\ \wedge\ \tau_2 = \{cPtr\} \cup \tau_{\text{left}} \cup \tau_{\text{right}}\\ \wedge\ \tau_2 \subseteq \sigma_2\end{array}\right\}$$

Finally, we drop unrequired pure constraints from the postcondition, leaving us with:

$$\left\{\begin{array}{l}\exists a, cPtr, \tau_{\text{left}}, \tau_{\text{right}}.\\ \quad modlist \mapsto a\\ \star\ CLseg(a, \mathsf{null}, \sigma_1)\\ \star\ tree \mapsto \text{opLbl}, cPtr, left, right, \_\\ \star\ \text{tree}(left, \tau_{\text{left}})\\ \star\ \text{tree}(right, \tau_{\text{right}})\\ \star\ Stuff\\ \star\ leftValaddr \mapsto \_\\ \wedge\ \tau_1 \subseteq \sigma_1\\ \wedge\ \tau_1 = \{cPtr\} \cup \tau_{\text{left}} \cup \tau_{\text{right}}\end{array}\right\}\ C\ \left\{\begin{array}{l}\exists a, \sigma_2, \tau_2, cPtr, \tau_{\text{left}}, \tau_{\text{right}}.\\ \quad modlist \mapsto a\\ \star\ CLseg(a, \mathsf{null}, \sigma_2)\\ \star\ tree \mapsto \text{opLbl}, cPtr, left, right, \_\\ \star\ \text{tree}(left, \tau_{\text{left}})\\ \star\ \text{tree}(right, \tau_{\text{right}})\\ \star\ Stuff\\ \star\ leftValaddr \mapsto \_\\ \wedge\ \sigma_1 \subseteq \sigma_2\\ \wedge\ \tau_2 \subseteq \sigma_2\\ \wedge\ \tau_2 = \{cPtr\} \cup \tau_{\text{left}} \cup \tau_{\text{right}}\end{array}\right\}$$

The final step to obtain the RHS of our implication (A.3) is to use (Mu) to unfold the *EvalCode* predicate in the precondition.

The proof for (A.2) is very similar to that for (A.1).

References

[1] B. Henderson, Linux loadable kernel module HOWTO (v1.09), 2006. Available online. `http://tldp.org/HOWTO/Module-HOWTO/`.

[2] G. Stoyle, M. Hicks, G. Bierman, P. Sewell, I. Neamtiu, Mutatis mutandis: Safe and predictable dynamic software updating, ACM Trans. Program. Lang. Syst. 29 (2007).

[3] I. Neamtiu, M. W. Hicks, G. Stoyle, M. Oriol, Practical dynamic software updating for C, in: M. I. Schwartzbach, T. Ball (Eds.), Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation (PLDI), Ottawa, Ontario, Canada, June 11-14, ACM, 2006, pp. 72–83.

[4] Z. Ni, Z. Shao, Certified assembly programming with embedded code pointers, in: J. G. Morrisett, S. L. Peyton-Jones (Eds.), Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006, ACM, 2006, pp. 320–333.

[5] P. J. Landin, The mechanical evaluation of expressions, Computer Journal 6 (1964) 308–320.

[6] E. Gamma, R. Helm, R. E. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, Reading, MA, 1995.

[7] K. Honda, N. Yoshida, M. Berger, An observationally complete program logic for imperative higher-order functions, in: 20th IEEE Symposium on Logic in Computer Science (LICS 2005), 26-29 June 2005, Chicago, IL, USA, IEEE Computer Society, 2005, pp. 270–279.

[8] L. Birkedal, B. Reus, J. Schwinghammer, H. Yang, A simple model of separation logic for higher-order store, in: L. Aceto, I. Damgård, L. A. Goldberg, M. M. Halldórsson, A. Ingólfsdóttir, I. Walukiewicz (Eds.), Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part II - Track B: Logic, Semantics, and Theory of Programming & Track C: Security and Cryptography Foundations, volume 5126 of *Lecture Notes in Computer Science*, Springer, 2008, pp. 348–360.

[9] J. Schwinghammer, L. Birkedal, B. Reus, H. Yang, Nested Hoare triples and frame rules for higher-order store, in: E. Grädel, R. Kahle (Eds.), Computer Science Logic, 23rd international Workshop, CSL 2009, 18th Annual Conference of the EACSL, Coimbra, Portugal, September 7-11, 2009, volume 5771 of *Lecture Notes in Computer Science*, Springer, 2009, pp. 440–454.

[10] N. Charlton, B. Horsfall, B. Reus, Crowfoot: A verifier for higher-order store programs, in: V. Kuncak, A. Rybalchenko (Eds.), VMCAI, volume 7148 of *Lecture Notes in Computer Science*, Springer, 2012, pp. 136–151.

[11] D. Keppel, S. J. Eggers, R. R. Henry, A Case for Runtime Code Generation, Technical Report UWCSE 91-11-04, University of Washington Department of Computer Science and Engineering, 1991.

[12] J. Schwinghammer, L. Birkedal, B. Reus, H. Yang, Nested Hoare triples and frame rule for higher-order store, Logical Methods in Computer Science 7 (2011).

[13] N. Charlton, B. Reus, Specification patterns and proofs for recursion through the store, in: O. Owe, M. Steffen, J. A. Telle (Eds.), Fundamentals of Computation Theory - 18th International Symposium, FCT 2011, Oslo, Norway, August 22-25, 2011, volume 6914 of *Lecture Notes in Computer Science*, Springer, 2011, pp. 310–321.

[14] R. Davies, F. Pfenning, A modal analysis of staged computation, J. ACM 48 (2001) 555–604.

[15] J. C. Reynolds, Separation logic: A logic for shared mutable data structures, in: 17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, IEEE Computer Society, 2002, pp. 55–74.

[16] B. Reus, T. Streicher, About Hoare logics for higher-order store, in: L. Caires, G. F. Italiano, L. Monteiro, C. Palamidessi, M. Yung (Eds.),

Automata, Languages and Programming, 32nd International Colloquium, ICALP 2005, Lisbon, Portugal, July 11-15, 2005, volume 3580 of *Lecture Notes in Computer Science*, Springer, 2005, pp. 1337–1348.

[17] L. Birkedal, N. Torp-Smith, H. Yang, Semantics of separation-logic typing and higher-order frame rules for Algol-like languages, LMCS 2 (2006).

[18] N. Charlton, B. Horsfall, B. Reus, Formal reasoning about runtime code update, in: S. Abiteboul, K. Böhm, C. Koch, K.-L. Tan (Eds.), ICDE Workshops, IEEE, 2011, pp. 134–138.

[19] M. Abadi, L. Cardelli, A Theory of Objects, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.

[20] J. Corbet, A. Rubini, G. Kroah-Hartman, Linux device drivers (third edition), O'Reilly Media, 2005.

[21] P. America, J. J. M. M. Rutten, Solving reflexive domain equations in a category of complete metric spaces, J. Comput. Syst. Sci. 39 (1989) 343–375.

[22] L. Birkedal, K. Støvring, J. Thamsborg, Realisability semantics of parametric polymorphism, general references and recursive types, Mathematical Structures in Computer Science 20 (2010) 655–703.

[23] M. Abadi, K. R. M. Leino, A logic of object-oriented programs, in: N. Dershowitz (Ed.), Verification: Theory and Practice. Essays Dedicated to Zohar Manna on the Occasion of his 64th Birthday, Lecture Notes in Computer Science, Springer, 2004, pp. 11–41.

[24] L. Birkedal, B. Reus, J. Schwinghammer, K. Støvring, J. Thamsborg, H. Yang, Step-indexed Kripke models over recursive worlds, in: T. Ball, M. Sagiv (Eds.), Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011, ACM, 2011, pp. 119–132.

[25] H. Cai, Z. Shao, A. Vaynberg, Certified self-modifying code, in: J. Ferrante, K. S. McKinley (Eds.), Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI), San Diego, California, USA, June 10-13, 2007, ACM, 2007, pp. 66–77.

[26] N. Charlton, Hoare logic for higher order store using simple semantics, in: L. D. Beklemishev, R. de Queiroz (Eds.), Logic, Language, Information and Computation - 18th International Workshop, WoLLIC 2011, Philadelphia, PA, USA, May 18-20, 2011, volume 6642 of *Lecture Notes in Computer Science*, Springer, 2011, pp. 52–66.

[27] B. Jacobs, J. Smans, F. Piessens, Verification of unloadable modules, in: M. Butler, W. Schulte (Eds.), FM 2011: Formal Methods - 17th International Symposium on Formal Methods, Limerick, Ireland, June 20-24,

2011, volume 6664 of *Lecture Notes in Computer Science*, Springer, 2011, pp. 402–416.

[28] N. Benton, Abstracting allocation, in: Z. Ésik (Ed.), Computer Science Logic, 20th International Workshop, CSL 2006, 15th Annual Conference of the EACSL, Szeged, Hungary, September 25-29, 2006, volume 4207 of *Lecture Notes in Computer Science*, Springer, 2006, pp. 182–196.

[29] K. Svendsen, L. Birkedal, M. Parkinson, Verifying generics and delegates, in: T. D'Hondt (Ed.), ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010, volume 6183 of *Lecture Notes in Computer Science*, Springer, 2010, pp. 175–199.

[30] A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, L. Birkedal, Ynot: dependent types for imperative programs, in: J. Hook, P. Thiemann (Eds.), Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008, ACM, 2008, pp. 229–240.

[31] A. Nanevski, J. G. Morrisett, L. Birkedal, Hoare type theory, polymorphism and separation, J. Funct. Program. 18 (2008) 865–911.

[32] R. L. Petersen, L. Birkedal, A. Nanevski, G. Morrisett, A realizability model for impredicative Hoare type theory, in: S. Drossopoulou (Ed.), Programming Languages and Systems, 17th European Symposium on Programming, ESOP 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008, volume 4960 of *Lecture Notes in Computer Science*, Springer, 2008, pp. 337–352.

[33] J. Berdine, C. Calcagno, P. W. O'Hearn, Smallfoot: Modular automatic assertion checking with separation logic, in: F. S. de Boer, M. M. Bonsangue, S. Graf, W. P. de Roever (Eds.), Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures, volume 4111 of *Lecture Notes in Computer Science*, Springer, 2005, pp. 115–137.

[34] D. Distefano, M. J. Parkinson, jStar: towards practical verification for Java, in: G. E. Harris (Ed.), Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-23, 2008, Nashville, TN, USA, ACM, 2008, pp. 213–226.

[35] B. Jacobs, J. Smans, F. Piessens, A quick tour of the VeriFast program verifier, in: K. Ueda (Ed.), Programming Languages and Systems - 8th Asian Symposium, APLAS 2010, Shanghai, China, November 28 - December 1, 2010, volume 6461 of *Lecture Notes in Computer Science*, Springer, 2010, pp. 304–311.

[36] B. Horsfall, N. Charlton, B. Reus, Verifying the reflective visitor pattern, in: Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs (FTfJP), Beijing, China, ACM, 2012, pp. 27–34.

[37] The Crowfoot website, 2012. `www.sussex.ac.uk/informatics/crowfoot`.

[38] J. Schwinghammer, H. Yang, L. Birkedal, F. Pottier, B. Reus, A semantic foundation for hidden state, in: L. Ong (Ed.), Foundations of Software Science and Computational Structures, 13th International Conference, FOSSACS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010, volume 6014 of *Lecture Notes in Computer Science*, Springer, 2010, pp. 2–17.