Specification patterns and proofs for recursion through the store

Nathaniel Charlton and Bernhard Reus

University of Sussex, Brighton

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 re-configured at runtime. Specifying such programs is challenging because of *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. 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.

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*.

This important language feature is the basis of flexible software systems that can be changed or re-configured 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 [9]. 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 [19, 14] 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 [15]). Semantically, 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" [13], 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 [8] which involves a mutual recursion between the visitor's methods and the visited structure's methods.

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

- We present and classify patterns of formal specification for programs that recurse through the store, using recursive assertions and nested triples (Section 4) ([18] considered only a very simple form for specifications).
- We state a generic "master pattern" covering all the recursively defined specification patterns we identified in this paper, and argue that the fixed points needed to give semantics to such specifications always exist (Section 4).
- 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 5). This is the first larger case study conducted with logical techniques based on [18], and shows that they are practical. Note that we use a slight extension of [18], adding call-by-value procedure parameters, inductively defined predicates and certain recursively defined families of assertions for lists. There is no space for giving proofs, but it should be pointed out that proofs have been done and we have developed a verification tool for support. We refer to this in the Conclusions (Section 6).

2 Our running example program

We now present an 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. Our program performs evaluation of (binary) expressions represented as binary trees. A tree node is either an integer leaf or a labeled binary fork which identifies the expression operator. The distinction is effected by a label which is 0 for operators and 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 tree node a pointer to the evaluation procedure to be used to evaluate it. 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.

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.

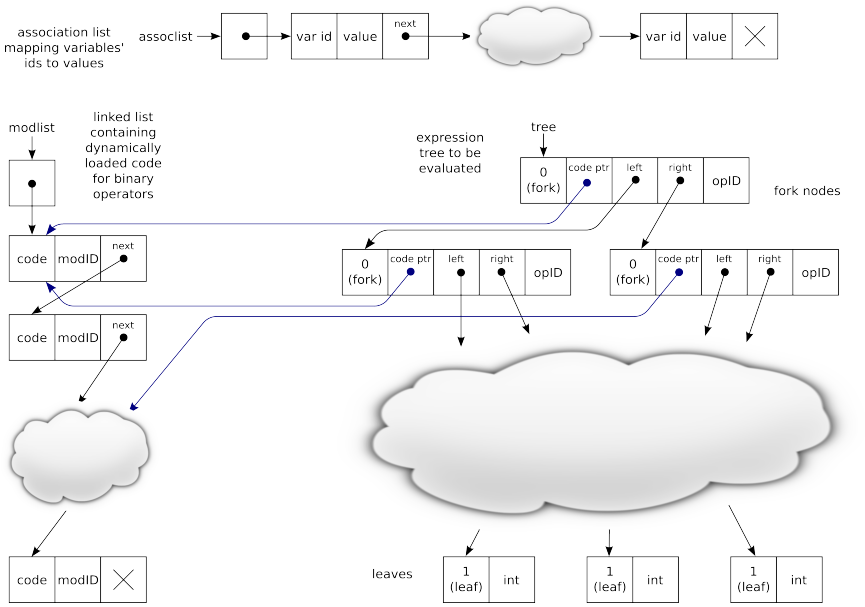

Fig. 1. The data structures used by our example program

The main program. Its code is given in Fig. 2. The eval $[e](\boldsymbol{e})$ statement invokes the code stored in the heap at address e, with a vector of (value) parameters $\boldsymbol{e}$. The shorthand res (explained later) simulates reference parameters. The expression '$\lambda\boldsymbol{x}.C$' denotes an *unevaluated* procedure with body C, taking formal parameters $\boldsymbol{x}$, as a value; thus $[e] := \text{'}\lambda\boldsymbol{x}.C\text{'}$ 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, their values do 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. In case the expression trees also encode variables, an association list mapping those variables to values is globally available at address *assoclist*.
2. Module-loading infrastructure: a linked list storing modules, pointed to by a globally known pointer *modlist*, and two procedures pointed to by *searchMods* and *loader*. Calling eval $[searchMods](opID, \mathsf{res}\ codeaddr)$ searches the list of loaded modules for the one implementing operator *opID*, returning its address or null (0) if it is not present. Calling eval $[loader](opID, \mathsf{res}\ codeaddr)$ always *guarantees* a module implementing operator *opID* is loaded, loading it if necessary, and returning its address.

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

```
[evalTree] :=
 ‘λ tree, resaddr.
  let kind = [tree] in
   if kind = 1 then
    let val = [tree + ValO] in [resaddr] := val
   else
    let codePtr = [tree + CodePtrO] in
     eval [codePtr](tree, resaddr)
 ’ ;
eval [evalTree](tree, res res)
```

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

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. For space reasons 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 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 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 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.

```
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’

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

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 ’ ’
```

Fig. 3. Code for some modules demonstrating various uses of higher order store

We finish by examining some further modules (also in Fig. 3) which illustrate more complex uses of higher-order store. The LOAD_OVERWRITE procedure first loads the code for the tree node's opID into the module list, then updates its own code pointer before calling that to evaluate the tree with the freshly loaded procedure. Note that next time the same code pointer in the tree is visited the newly loaded procedure is executed straight away and no more loading occurs. This update affects only the pointer in the tree data structure. 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 have also considered *specialisation of code at runtime*. We wrote an implementation of the binomial coefficient $\binom{n}{k} := n!/k!(n-k)!$ which, when it detects that its left subtree (i.e. n) is an integer literal, calculates $n!$ (once) and generates on the fly an optimised implementation of $\binom{n}{k}$ that reuses the value.

3 The programming and assertion languages

The programming language. We work with a simple imperative programming language extended with operations for stored procedures and heap ma-

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

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

$$\begin{aligned} P ::= \quad & \mathrm{True} \mid \mathrm{False} \mid P_1 \vee P_2 \mid P_1 \wedge P_2 \mid P_1 \Rightarrow P_2 \mid \forall x.P \mid \exists x.P \mid e_1 = e_2 \mid e_1 \leq e_2 \\ & \mid e_1 \mapsto e_2 \mid \mathsf{emp} \mid P_1 \star P_2 \mid \mathrm{lseg}(e_1, e_2, \Sigma) \mid \mathrm{tree}(e_1, \Sigma) \mid \ldots \\ & \mid \{P\}\, e(\boldsymbol{e})\, \{Q\} \mid P \otimes Q \mid \Sigma_1 \subseteq \Sigma_2 \mid (\mu^k\ R_1(\boldsymbol{x}_1), \ldots, R_n(\boldsymbol{x}_n)\ .\ \mathscr{A}_1, \ldots, \mathscr{A}_n)(\boldsymbol{e}) \end{aligned}$$

Fig. 4. Syntax of expressions, commands and assertions

nipulation, as described and used in Section 2. The syntax of the language is shown in Fig. 4, where $\boldsymbol{x}$ (resp. $\boldsymbol{e}$) represents a vector of distinct variables (resp. a vector of expressions). This language extends that in [18] 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 $\mathsf{eval}\ [e](\boldsymbol{e}, \mathsf{res}\ v)\ ;\ C$ as shorthand for $\mathsf{let}\ \mathit{vaddr} = \mathsf{new}\ 0\ \mathsf{in}\ (\mathsf{eval}\ [e](\boldsymbol{e}, \mathit{vaddr})\ ;\ \mathsf{let}\ v = [\mathit{vaddr}]\ \mathsf{in}\ C\ ;\ \mathsf{free}\ \mathit{vaddr})$.

The assertion language. The assertion language, shown in Fig. 4, follows [18], adding finite sets, more general recursive definitions and inductive predicates, and with some changes to accommodate parameter passing. The language is based on first-order intuitionistic logic augmented with the standard connectives of separation logic [17], and several further extensions:

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. The triple $\{P\}\, e(\boldsymbol{e})\, \{Q\}$ means that e denotes code satisfying $\{P\} \cdot \{Q\}$ when invoked with parameters $\boldsymbol{e}$. For code that does not expect any parameters, $\boldsymbol{e}$ will have length zero and we write simply $\{P\}\, e\, \{Q\}$.

Invariant extension: Intuitively invariant extension $P \otimes Q$ denotes a modification of P where all the pre- and post-conditions of triples inside P are $\star$-extended with Q. The operator $\otimes$ is from [4, 18] and is not symmetric.

Inductively defined predicates: Predicates describing the linked lists and trees we use are available; their defining axioms are given in Fig. 5. The "..." in Fig. 4 indicates that any similar predicates can be added as required. $\mathrm{lseg}(x, y, \Sigma)$ denotes a linked list segment from x to y, of nodes of three cells each, where Σ is the set of addresses of the nodes. $\mathrm{lseg}{<}T(\cdot){>}(x, y, \Sigma)$ says additionally that the first value in each of the segment's nodes satisfies T, which is an assertion with an expression hole. $\mathrm{tree}(t, \Sigma)$ denotes an expression tree rooted at t where the code pointers pointing *out of* the tree point to the set of addresses Σ.

(Mutually) Recursively defined assertions: Recursively defined assertions are the key to our work, because they let us reason naturally about chal-

$$\mathrm{lseg}(x,y,\sigma) \;\Leftrightarrow\; (x = y \wedge \sigma = \emptyset \wedge \mathsf{emp}) \vee (\exists nxt, \sigma' \,.\, x \mapsto \_, \_, nxt \star \mathrm{lseg}(nxt, y, \sigma') \wedge nxt \neq x \wedge \sigma = \sigma' \cup \{x\})$$

$$\mathrm{lseg}\,\langle T(\cdot)\rangle\,(x,y,\sigma) \;\Leftrightarrow\; \mathrm{lseg}(x,y,\sigma) \wedge \forall a.a \in \sigma \Rightarrow (a \mapsto T(\cdot) \star \mathrm{True})$$

$$\mathrm{tree}(t,\tau) \;\Leftrightarrow\; \mathrm{tree}_{\mathrm{fork}}(t,\tau) \vee (\exists n \,.\, t \mapsto 1, n \wedge \tau = \emptyset)$$

$$\mathrm{tree}_{\mathrm{fork}}(t,\tau) \;\Leftrightarrow\; \exists \mathit{codePtr}, \mathit{left}, \mathit{right}, \mathit{opId}, \tau', \tau''. \quad \tau = \{\mathit{codePtr}\} \cup \tau' \cup \tau'' \wedge t \mapsto 0, \mathit{codePtr}, \mathit{left}, \mathit{right}, \mathit{opId} \star \mathrm{tree}(\mathit{left}, \tau') \star \mathrm{tree}(\mathit{right}, \tau'')$$

Fig. 5. Inductively defined predicates used to specify and prove our example program.

lenging patterns of execution, such as self-updating code and recursion through the store. We use the notation $\mu^k\ R_1(\boldsymbol{x}_1), \ldots, R_n(\boldsymbol{x}_n)\ .\ P_1, \ldots, P_n$ to indicate that n predicates are defined mutually recursively with arguments $\boldsymbol{x}_i$ and bodies P_i, respectively, and the superscript k indicates that the k-th such predicate is selected. Note that (just to save space) we avoid introducing a syntactic category for predicates so $(\mu^k\ R_1(\boldsymbol{x}_1), \ldots, R_n(\boldsymbol{x}_n).P_1, \ldots, P_n)(\boldsymbol{e})$ is a formula but without arguments $\mu^k\ R_1(\boldsymbol{x}_1), \ldots, R_n(\boldsymbol{x}_n).P_1, \ldots, P_n$ is not a syntactically correct construct. Throughout the paper we will, however, for the sake of brevity, use abbreviations of the form $A := \mu^k\ R_1(\boldsymbol{x}_1), \ldots, R_n(\boldsymbol{x}_n).P_1, \ldots, P_n$ that are understood to be used with proper arguments in formulae. In Section 4 we will give a grammar for formulae that can be allowed in those recursive definitions since existence of fixed points is not automatic. We write $\mathscr{A}$ for an *allowed formula*, i.e. one that is of an appropriate form to ensure the existence of a solution; recursion variables R_i are only allowed to appear inside such an $\mathscr{A}$.

Each assertion describes a property of states, which consist of an (immutable) environment and a mutable heap. 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 $e \mapsto e_0, \ldots, e_n := e \mapsto e_0 \star \cdots \star (e{+}n) \mapsto e_n$. The set of free variables of an expression or assertion is largely obvious, but note that $fv(\text{`}\lambda\boldsymbol{x}.C\text{'}) := fv(C) - \boldsymbol{x}$.

4 Specification patterns for recursion through the store

In this section we present a series of patterns, of increasing complexity, for specifying recursion through the store. 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 specification Φ_1 in Fig. 6 describes code that operates on a data structure D and calls itself recursively through a pointer g into the heap. Φ_2 (also in Fig. 6) describes two

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

$\Phi_2 := \quad \mu^1 R \,.\, g_1 \mapsto \forall \boldsymbol{x}_1.\{D_1 \star \boxed{R}\} \cdot (\boldsymbol{p}_1)\{D_2 \star \boxed{R}\} \star g_2 \mapsto \forall \boldsymbol{x}_2.\{D_3 \star \boxed{R}\} \cdot (\boldsymbol{p}_2)\{D_4 \star \boxed{R}\}$

With dynamic loader: $\Phi_{\text{loader}} := \quad R_{\text{loaded}}(g_1) \star R_{\text{loaded}}(g_2) \star \mathit{LoaderCode}$ where

$R_{\text{loaded}} := \quad \mu^1\ R(c), \mathit{LoaderCode}\ .$

$c \mapsto \forall \boldsymbol{x}.\{D \star \boxed{R(g_1)} \star \boxed{R(g_2)} \star \boxed{\mathit{LoaderCode}}\} \cdot (\boldsymbol{p})\{D \star \boxed{R(g_1)} \star \boxed{R(g_2)} \star \boxed{\mathit{LoaderCode}}\},$

$\mathit{loader} \mapsto \forall a, \mathit{ID}.\ \{a \mapsto \_\} \cdot (a, \mathit{ID})\,\{\boxed{R(a)}\}$

List of code: $\mathit{CLseg} := \quad \mu^1 R(x, y, \sigma)\ .\ \text{lseg}\langle \boxed{T_0(\cdot)} \rangle (x, y, \sigma) \qquad$ where $T_0(\cdot)$ is

$\forall \boldsymbol{x}.\{\exists a, \sigma.\ D \star \mathit{codelist} \mapsto a \star \boxed{R(a, \mathsf{null}, \sigma)}\} \cdot (\boldsymbol{p})\{\exists a, \sigma.\ D \star \mathit{codelist} \mapsto a \star \boxed{R(a, \mathsf{null}, \sigma)}\}$

Data structure with pointers: $\mathit{CLseg}' := \ \mu^1 R(x, y, \sigma).\ \text{lseg}\langle \boxed{T_1(\cdot)} \rangle (x, y, \sigma)$ where $T_1(\cdot)$ is

$$\forall \boldsymbol{x}\ .\ \left\{ \begin{array}{l} \exists a, \sigma, \tau.\ \tau \subseteq \sigma\ \wedge\ D(\tau) \\ \star\ \mathit{codelist} \mapsto a\ \star\ \boxed{R(a, \mathsf{null}, \sigma)} \end{array} \right\} \cdot (\boldsymbol{p}) \left\{ \begin{array}{l} \exists a, \sigma, \tau.\ \tau \subseteq \sigma\ \wedge\ D(\tau) \\ \star\ \mathit{codelist} \mapsto a\ \star\ \boxed{R(a, \mathsf{null}, \sigma)} \end{array} \right\}$$

The master pattern: $\mu^k\ R_1(\boldsymbol{x}_1), \ldots, R_n(\boldsymbol{x}_n)\ .\ \mathscr{A}_1, \ldots, \mathscr{A}_n$

where the following grammar shows what form each $\mathscr{A}_i$ can take (here $i_1, \ldots, i_m$ and $i'_1, \ldots, i'_{m'}$ are in $\{1, \ldots, n\}$, and P, P' are formulae not containing any of $R_1, \ldots, R_n$):

$$S ::= \ \forall \boldsymbol{x}_1.\ \{\exists \boldsymbol{x}_2\ .\ R_{i_1}(\boldsymbol{e}_1) \star \cdots \star R_{i_m}(\boldsymbol{e}_m) \star P\} \cdot (\boldsymbol{x}_3)\ \{\exists \boldsymbol{x}_4\ .\ R_{i'_1}(\boldsymbol{e}'_1) \star \cdots \star R_{i'_{m'}}(\boldsymbol{e}'_{m'}) \star P'\}$$

$$T ::= \quad t \mapsto S(\cdot) \mid t \mapsto x \wedge S(x) \qquad \mathscr{A}_i ::= \quad T_1 \star \cdots \star T_k \mid \text{lseg}\langle S \rangle (\boldsymbol{x})$$

Fig. 6. Specification patterns for recursion through the store.

pieces of code on the heap that call themselves and each other recursively via two pointers g_1 and g_2. Similarly $\Phi_3, \Phi_4, \ldots$ can be formulated. (The $D, D_1, D_2, \ldots$ in Fig. 6 are metavariables, and in applications of the patterns they will be replaced by concrete formulae describing data structures.)

Specification Φ_1 allows the code pointed to by g to update itself (like our OSCILLATE example). Similarly, specification Φ_2 allows pieces of code in g_1 and g_2 to update themselves and additionally to update each other. Such updates are permitted as long as the update is with code that behaves in the same way[1][2].

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 function-

[1] Unlike the types used, say, in [2], our nested triples can handle updates that do not preserve code behaviour; examples in this paper do not use such updates, however.

[2] There are 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.

ally specified using the following instance of the Φ_1 pattern:

$$\mu^1 R \,.\, g \mapsto \forall x, n.\{x \mapsto n \star r \mapsto \_ \star \boxed{R}\} \cdot (x)\{x \mapsto 0 \star r \mapsto n! \star \boxed{R}\}$$

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 Φ_{loader} in Fig. 6 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 $\otimes$: R appears in the postcondition but nowhere in the precondition. Note also that we have omitted the analogous definition of *LoaderCode* (using μ^2) here for brevity.

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 at *codelist* contains a pointer to the start of the list. To reason about this setup, we use lseg<T> (Fig. 5) to define recursively a predicate *CLseg* (Fig. 6) for segments of code lists. Note the existential quantifiers over a and σ in the auxiliary T_0: these mean 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 the new code also behaves again in the same way. One can constrain this behaviour by varying the specification in several ways; for instance, we can allow the pieces of code in the list to call each other but prohibit them from updating each other.

We point out the similarity between our idealised code lists and for example the *net_device* list that the Linux kernel uses to manage dynamically loaded and unloaded network device drivers [6].

Recursion via a set of pointers stored in a data structure. Instead of finding the right code to call explicitly within a list of type *CLseg* (using the ID numbers) 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*$'$ (in Fig. 6); the constraint $\tau \subseteq \sigma$ says that all code pointers in D must point into the code list. Our example program combines this kind of recursion through the store with use of a loader function; we will see the specifications needed in Section 5 where the $D(\cdot)$ will be tree$(a, \cdot)$.

The master pattern. The last part of Fig. 6 presents a master pattern, which encompasses all the specification patterns seen so far. We have shown that recursive definitions of this form admit unique solutions; due to space constraints the proof of this, which uses the model and techniques of [18], is omitted. The $\otimes$ operator does not appear in the master pattern, but its effect can be expressed by unfolding its definition using the distributions axioms for $\otimes$ as found in [18].

$$
\begin{array}{l}
\mathit{CLseg} \quad := \quad \mu^1\ \mathit{CLseg}(x, y, \sigma), \mathit{EvalCode}, \mathit{LoaderCode}, \mathit{SearchModsCode}\ . \\
\quad \mathrm{lseg}{<}\ \mathit{Mod}\ {>}(x, y, \sigma), \\
\quad \mathit{evalTree} \mapsto\ \mathit{EvalTriple}(\cdot)\ , \\
\quad \mathit{loader} \mapsto \forall \mathit{opID}, \mathit{codePtraddr}, \sigma_1\ . \\
\qquad \left\{ \begin{array}{l} \exists a\ . \\ \quad \mathit{modlist} \mapsto a \\ \star\ \mathit{CLseg}(a, \mathsf{null}, \sigma_1) \\ \star\ \mathit{codePtraddr} \mapsto \_ \end{array} \right\} \cdot (\mathit{opID}, \mathit{codePtraddr}) \left\{ \begin{array}{l} \exists a, r, \sigma_2.\quad \sigma_1 \cup \{r\} \subseteq \sigma_2 \wedge \\ \quad \mathit{modlist} \mapsto a \\ \star\ \mathit{CLseg}(a, \mathsf{null}, \sigma_2) \\ \star\ \mathit{codePtraddr} \mapsto r \end{array} \right\}, \\
\quad \mathit{searchMods} \mapsto \forall \mathit{opID}, \mathit{codePtraddr}, a, \sigma\ . \\
\qquad \left\{ \begin{array}{l} \quad \mathit{modlist} \mapsto a \\ \star\ \mathit{CLseg}(a, \mathsf{null}, \sigma) \\ \star\ \mathit{codePtraddr} \mapsto \_ \end{array} \right\} \cdot (\mathit{opID}, \mathit{codePtraddr}) \left\{ \begin{array}{l} \exists r.\quad (r \in \sigma \vee r = \mathsf{null}) \wedge \\ \quad \mathit{modlist} \mapsto a \\ \star\ \mathit{CLseg}(a, \mathsf{null}, \sigma) \\ \star\ \mathit{codePtraddr} \mapsto r \end{array} \right\}
\end{array}
$$

where $\mathit{Mod}(F)$ abbreviates

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

and $\mathit{EvalTriple}(F)$ is the same, but with tree in place of $\mathrm{tree}_{\mathrm{fork}}$ in the precondition.

Fig. 7. Definitions of predicates used in the specification of our example program.

5 Specifying and proving our example program

In this section we show how to specify memory safety of our example program; due to space restrictions we can only hint briefly at the proof. Our program uses three heap data structures (Fig. 1) as well as various procedures stored on the heap; Fig. 7 defines the predicates we use to describe these. All uses of μ fit the master pattern. By convention predicates named *Proc*Code assert that the variable *proc* points to some appropriate code stored in the heap. The abbreviation $\mathit{Mod}(F)$ used here says that code F behaves appropriately to be used as a module in our system. $\mathit{Mod}(F)$ is almost the same as the specification for *evalTree*; the difference is that modules may assume the tree is a fork, because *evalTree* has already checked for and handled the leaf case.

In the proof obligation of the main program (Fig. 2) we *assume* specifications for the procedures *loader* and *searchMods* (for which the code is not given):

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

A more specific post-condition could be used to prove the absence of memory leaks etc., but True is sufficient for memory safety. For the proof we use some obvious adaptations of rules as presented in [18].

We remark that because the main program doesn't manipulate the association list at all, we can omit *AssocList* from our proof of the main program. Once our proof is complete, we can use the so-called *deep frame rule* ($\otimes$–FRAME in [18]) to add *AssocList* everywhere it is needed. To prove memory safety of the modules we must show for each module implementation M that $\mathit{Mod}(M) \otimes \mathit{AssocList}$. For modules that do not directly access the association list (e.g. PLUS), we first prove $\mathit{Mod}(M)$ and then use $\otimes$–FRAME to add *AssocList*. For modules that do access the association list, e.g. VAR and ASSIGN, one must prove $\mathit{Mod}(M) \otimes \mathit{AssocList}$ directly.

The proofs for the modules are very similar to that for the main program. One difference is that one must apply the $\star$-FRAME axiom to nested triples as well: e.g. in the module PLUS, the first eval works only on the left subtree, so we use $\star$-FRAME (see [18]) to add the root node and the right subtree as invariants to the triple for *evalTree*. Here we see 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* code from the code list.

6 Conclusions and future work

We extended the separation logic of [18] 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.

The work most closely related to ours (other than [18] on which we build) is that by Honda *et al.* [10], which also provides a Hoare logic with nested triples. It discusses the proof of a factorial function which calls itself through the store, but does not consider more complex patterns of recursion through the store. In [10] *content quantification* is used instead of separation logic, consequently ignoring frame rules, and total rather than partial correctness is treated.

We plan to extend our language and logic with (extensional) operations for generating code at runtime. We will also study the relationship between nested triples and the specifications based on *abstract predicate families* used in [16].

Finally, the complexity of the involved specifications and proofs demands tool support. We have developed a verification tool *Crowfoot* [1] supporting a logic and a language very similar to that used in this paper. We have used *Crowfoot*, for example, to verify correctness of runtime module updates [5], and to verify a

version of the example presented in this paper. Use of $\star$-FRAME by *Crowfoot* is automatic, whereas the use of $\otimes$-FRAME is presently guided by user annotations. Our tool has been inspired by tools for separation logic like Smallfoot [3], jStar [7] and VeriFast [11] (a small survey of related work can be found in [12]).

Acknowledgements We would like to thank J. Schwinghammer, H. Yang, F. Pottier, and L. Birkedal and for discussions on the usage of recursive families of specifications. Our research has been supported by EPSRC grant EP/G003173/1.

References

1. The Crowfoot website (includes a version of the example in this paper) available at: www.informatics.sussex.ac.uk/research/projects/PL4HOStore/crowfoot/.
2. M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.
3. J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, pages 115–137, 2005.
4. L. Birkedal, N. Torp-Smith, and H. Yang. Semantics of separation-logic typing and higher-order frame rules for Algol-like languages. *LMCS*, 2(5), 2006.
5. N. Charlton, B. Horsfall, and B. Reus. Formal reasoning about runtime code update. In S. Abiteboul, K. Böhm, C. Koch, and K.-L. Tan, editors, *ICDE Workshops*, pages 134–138. IEEE, 2011.
6. J. Corbet, A. Rubini, and G. Kroah-Hartman. *Linux device drivers (third edition)*. O'Reilly Media, 2005.
7. D. Distefano and M. J. Parkinson. jStar: towards practical verification for Java. In *OOPSLA*, pages 213–226, 2008.
8. E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.
9. B. Henderson. Linux loadable kernel module HOWTO (v1.09), 2006. Available online. http://tldp.org/HOWTO/Module-HOWTO/.
10. K. Honda, N. Yoshida, and M. Berger. An observationally complete program logic for imperative higher-order functions. In *LICS*, pages 270–279, 2005.
11. B. Jacobs, J. Smans, and F. Piessens. A quick tour of the VeriFast program verifier. In *APLAS*, pages 304–311, 2010.
12. N. R. Krishnaswami, J. Aldrich, L. Birkedal, K. Svendsen, and A. Buisse. Design patterns in separation logic. In *TLDI*, pages 105–116, 2009.
13. P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320, Jan. 1964.
14. I. Neamtiu, M. W. Hicks, G. Stoyle, and M. Oriol. Practical dynamic software updating for C. In *PLDI*, pages 72–83, 2006.
15. Z. Ni and Z. Shao. Certified assembly programming with embedded code pointers. In *POPL*, pages 320–333, 2006.
16. M. J. Parkinson and G. M. Bierman. Separation logic, abstraction and inheritance. In *POPL*, pages 75–86, 2008.
17. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74, 2002.
18. J. Schwinghammer, L. Birkedal, B. Reus, and H. Yang. Nested Hoare triples and frame rules for higher-order store. In *CSL*, pages 440–454, 2009.
19. G. Stoyle, M. Hicks, G. Bierman, P. Sewell, and I. Neamtiu. Mutatis mutandis: Safe and predictable dynamic software updating. *ACM Trans. Program. Lang. Syst.*, 29(4), 2007.