Structured Specifications for Better Verification of Heap-Manipulating Programs

Cristian Gherghina[1] Cristina David[1] Shengchao Qin[2] Wei-Ngan Chin[1]

[1] Department of Computer Science, National University of Singapore
[2] School of Computing, University of Teesside

Abstract. Conventional specifications typically have a flat structure that is based primarily on the underlying logic. Such specifications lack structures that could have provided better guidance to the verification process. In this work, we propose to add three new structures to a specification framework for separation logic to achieve a *more precise* and *better guided* verification for pointer-based programs. The newly introduced structures empower users with more control over the verification process in the following ways: (i) case analysis can be invoked to take advantage of disjointness conditions in the logic. (ii) early, as opposed to late, instantiation can minimise on the use of existential quantification. (iii) formulae that are staged provide better reuse of the verification process.
Initial experiments have shown that structured specifications can lead to more precise verification without incurring any performance overhead.

1 Introduction

Recent developments of the specification mechanisms have focused mostly on expressiveness [2, 1, 5] (to support verification for more properties), abstraction [16, 18] (to support information hiding in specification) and modularity [14, 7, 8] (to support more readable and reusable specifications). To the best of our knowledge, there has been hardly any attempt on the development of specification mechanisms that could support better verifiability (in terms of both efficiency and effectiveness). Most efforts on better verifiability have been confined to the verification technology; an approach that may lead to less portability (as we become more reliant on clever heuristics from the verification tools) and also more complex implementation for the verification tools themselves. In this paper, we shall propose a novel approach towards better verifiability that focuses on new structures in the specification mechanism instead.

To illustrate the need for an enhanced specification mechanism, we will make use of separation logic, which allows for a precise description of heap-based data structures and their properties. As an example, consider a data node `node2` and a predicate describing an AVL tree that captures the size property via `s` and the height via `h`:

$$\texttt{data node2}\ \{\ \texttt{int val; int height; node2 right; node2 left;}\ \}$$
$$\texttt{avl}\langle \texttt{root}, \texttt{h}, \texttt{s}\rangle \equiv \texttt{root}{=}\texttt{null} \wedge \texttt{h}{=}0 \wedge \texttt{s}{=}0$$
$$\vee\ \texttt{root} \mapsto \texttt{node2}\langle \_, \texttt{h}, \texttt{r}, \texttt{l}\rangle * \texttt{avl}\langle \texttt{r}, \texttt{h}_1, \texttt{s}_1\rangle * \texttt{avl}\langle \texttt{l}, \texttt{h}_2, \texttt{s}_2\rangle \wedge \texttt{h} = \texttt{max}(\texttt{h}_1, \texttt{h}_2){+}1$$
$$\wedge -1 {\leq} \texttt{h}_1 {-} \texttt{h}_2 {\leq} 1 \wedge \texttt{s}{=}\texttt{s}_1{+}\texttt{s}_2{+}1$$

Formula $\mathtt{p}\mapsto\mathtt{c}\langle\mathtt{v}^*\rangle$ denotes a points-to fact of the heap where c is a data node with $\mathtt{v}^*$ as its arguments, while spatial conjunction $\Phi_1{*}\Phi_2$ denotes a program state with two disjoint heap spaces described by sub-formulae Φ_1 and Φ_2, respectively. These two notations of separation logic allow heap states to be expressed in a succinct manner.

The aforementioned definition asserts that an AVL tree is either empty (the base case $\mathtt{root}{=}\mathtt{null}\wedge\mathtt{h}{=}0\wedge\mathtt{s}{=}0$), or it consists of a data node ($\mathtt{root}\mapsto\mathtt{node2}\langle\_,\mathtt{h},\mathtt{r},\mathtt{l}\rangle$) and two disjoint subtrees ($\mathtt{avl}\langle\mathtt{r},\mathtt{h}_1,\mathtt{s}_1\rangle{*}\mathtt{avl}\langle\mathtt{l},\mathtt{h}_2,\mathtt{s}_2\rangle$). Each node is used to store the actual data in the val field, and the maximum height of the current subtree in the height field. The constraint $-1{\leq}\mathtt{h}_1{-}\mathtt{h}_2{\leq}1$ states that the tree is balanced, while $\mathtt{s}{=}\mathtt{s}_1{+}\mathtt{s}_2{+}1$ and $\mathtt{h}{=}\mathtt{max}(\mathtt{h}_1,\mathtt{h}_2){+}1$ compute the size and height of the tree pointed by root from the properties $\mathtt{s}_1,\mathtt{s}_2$ and $\mathtt{h}_1,\mathtt{h}_2$, respectively, that are obtained from the two subtrees. The $*$ connector ensures that the head node and the right and left subtrees reside in disjoint heaps. Our system automatically generates existential quantifiers for local values and pointers, such as $\mathtt{r},\mathtt{l},\mathtt{h}_1,\mathtt{h}_2,\mathtt{s}_1,\mathtt{s}_2$.

Next, we specify a method that attempts to retrieve the height information from the root node of the data structure received as argument. In case the argument has the value null, the method returns 0, as captured by $\mathtt{res}{=}0$. To provide a suitable link between pre- and post-conditions, we use the logical variables $\mathtt{v},\mathtt{h},\mathtt{lt},\mathtt{lr}$ that have to be instantiated for each call to the method. As a first try, we capture both the null and non-null scenarios as a composite formula consisting of a disjunction of the two cases, as shown below:

$$\begin{array}{l}\mathtt{int\ get\_height(node2\ x)}\\ \quad\mathtt{requires\ x{=}null}\vee\mathtt{x}\mapsto\mathtt{node2}\langle\mathtt{v},\mathtt{h},\mathtt{lt},\mathtt{lr}\rangle\\ \quad\mathtt{ensures\ (x{=}null}\wedge\mathtt{res{=}0)}\vee(\mathtt{x}\mapsto\mathtt{node2}\langle\mathtt{v},\mathtt{h},\mathtt{lt},\mathtt{lr}\rangle\wedge\mathtt{res{=}h});\\ \{\mathtt{if\ (x = null)\ then\ 0\ else\ x.height}\}\end{array}$$

This specification introduces disjunctions both in the pre and post-conditions, which would make the verification process perform search over the disjuncts[17]. Basically, each disjunct corresponds to an acceptable scenario of which at least one needs to be proven. However, there are situations when the program state does not contain enough information to determine which of the scenarios applies. For illustration, let us consider that we are interested in retrieving the height information for an AVL tree pointed by x and the program state before the call to the get_height method is $\mathtt{avl}\langle\mathtt{x},\mathtt{h}_1,\mathtt{s}_1\rangle$. We have to verify that the current program state obeys the method's precondition. However, when verifying the null and non-null scenarios separately, both checks fail as the program state $\mathtt{avl}\langle\mathtt{x},\mathtt{h}_1,\mathtt{s}_1\rangle$ does not contain sufficient information to conclude neither that $\mathtt{x}{\neq}\mathtt{null}$, nor that $\mathtt{x}{=}\mathtt{null}$. We provide the two failing verification conditions in the form of the entailment procedure from [17]: $\Phi_a\vdash\Phi_c*\Phi_r$, where the antecedent Φ_a and consequent Φ_c are given, while the residue Φ_r is to be computed. This entailment finds a subheap in Φ_a that satisfies Φ_c and returns the unused subheap from Φ_a as residue Φ_r. Getting back to the current get_height example, the two failing entailments are given below. As none of the following two entailments succeeds, the verification of the method call fails.

$$\begin{array}{c}\mathtt{avl}\langle\mathtt{x},\mathtt{h}_1,\mathtt{s}_1\rangle\vdash(\mathtt{x{=}null}){*}\Phi_{\mathtt{r}_1}\\ \mathtt{avl}\langle\mathtt{x},\mathtt{h}_1,\mathtt{s}_1\rangle\vdash(\mathtt{x}\mapsto\mathtt{node2}\langle\mathtt{v},\mathtt{h},\mathtt{lt},\mathtt{lr}\rangle){*}\Phi_{\mathtt{r}_2}\end{array}$$

As a second try, we write the specification in a modular fashion by separating the two scenarios as advocated by past works [14, 7]. In [14], Leavens and Baker proposed for each specification to be decomposed into multiple specifications (where it is called case analysis) to capture different scenarios of usage. Their goal was improving the readability of specifications, as smaller and simpler specifications are easier to understand than larger ones. In [7] multiple specifications were advocated to help achieve more scalable program verification. By using multiple pre/post conditions, we obtain the following specification:

$\texttt{int get\_height(node2 x)}$
$\texttt{requires x=null ensures res=0;}$
$\texttt{requires x}\mapsto\texttt{node2}\langle\texttt{v,h,lt,rt}\rangle\ \texttt{ensures x}\mapsto\texttt{node2}\langle\texttt{v,h,lt,rt}\rangle\wedge\texttt{res=h;}$

During the verification process, each scenario (denoted by a pre/post-condition pair) is proven separately [7]. However, neither of the two entailments (for each of the two scenarios) succeeds, causing the verification of the method call to fail.

A possible solution is to perform case analysis on variable x: first assume x=null, then assume x≠null, and try to prove both cases. For soundness, these cases must be disjoint and exhaustively cover all scenarios. Accordingly, the following two provable entailments are obtained, and the verification succeeds:

$$\texttt{avl}\langle\texttt{x},\texttt{h}_1,\texttt{s}_1\rangle\wedge\texttt{x=null}\vdash(\texttt{x=null})*\Phi_{\texttt{r}_1}$$
$$\texttt{avl}\langle\texttt{x},\texttt{h}_1,\texttt{s}_1\rangle\wedge\texttt{x}\neq\texttt{null}\vdash(\texttt{x}\mapsto\texttt{node2}\langle\texttt{v,h,lt,lr}\rangle)*\Phi_{\texttt{r}_2}$$

However, case analysis is not always available in provers, as it might be tricky to decide on the condition for a case split. Traditionally, the focus of specification mechanism has been on improving its ability to cover a wider range of problems more accurately, while the effectiveness of verification is left to the underlying provers. In this paper, we attempt a novel approach, where the focus is on determining a good specification mechanism to achieve better expressivity and verifiability.

Often, a user has an intuition about the proving process. In the current work, we provide the necessary utensils for integrating this intuition in the specification in order to guide the verification. Instead of writing a flat (unstructured) specification, the user can use insights about the proof for writing a structured specification that will trigger different techniques during the proving process:

– **Case analysis** is conventionally captured as part of the proving process. The user typically indicates the program location where case analysis is to be performed [23]. This corresponds to performing a case analysis on some program state (or antecedent) of the proving process. In our approach, we provide a case construct to distinguish the input states of pre/post specifications instead. This richer specification can be directly used to guide the verification process. For the aforementioned get_height method, the case structured specification will automatically force a case split on x:

$\texttt{case}\{\texttt{x=null}\ \rightarrow\ \texttt{ensures res=0;}$
$\texttt{x}\neq\texttt{null}\ \rightarrow\ \texttt{requires x}\mapsto\texttt{node2}\langle\texttt{v,h,lt,lr}\rangle$
$\texttt{ensures x}\mapsto\texttt{node2}\langle\texttt{v,h,lt,lr}\rangle\wedge\texttt{res=h}\};$

– **Early vs. late instantiations** denote different types of bindings for the logical variables (of consequent) during the entailment proving process. Early instantiation is an instantiation that occurs at the first occurrence of its logical variable, while late instantiation occurs at the last occurrence of its logical variable. While late instantiation can be more accurate for variables that are constructed from inequality constraints, early instantiation can typically be done with fewer existential quantifiers since instantiation converts these existential logical variables to quantifier-free form at an earlier point. We propose to use early instantiation, by default, and only to resort to late instantiation when explicitly requested by the programmer.
– **Staged formulae** allows the specification to be made more concise through sharing of common sub-formulae. Apart from better sharing, this also allows verification to be carried out incrementally over multiple (smaller) stages, instead of a single (larger) stage. The need for early/late instantiations, as well as for staged formulae will be motivated in more details later in Sec 2.

In the rest of the paper we shall focus on the apparatus for writing and verifying (or checking) structured specifications. Sec 2 provides examples to motivate the need for two other aspects of structured specifications. Sec 3 formalizes the notion of structured specifications. Sec 4 formalizes the verification rules to generate Hoare triples and entailment proving for structured specifications, while Sec 5 presents our experimental results before some concluding remarks in Sec 6.

2 Motivating Examples

In the current section we present two more examples that motivate our enhancements to the specification mechanism.

2.1 Example 1

Consider a method that receives two AVL trees, `t1` and `t2`, and merges them by recursively inserting all the elements of `t2` into `t1`. By using the case construct introduced in Sec 1 we may write a case structured specification, which captures information about the resulting tree size when `t1` is not `null`, and about the resulting size and height, whenever `t1` is `null`:

$$\begin{array}{l}\texttt{case}\{\texttt{t1} = \texttt{null} \rightarrow \texttt{requires avl}\langle\texttt{t2},\texttt{s2},\texttt{h2}\rangle\texttt{ensures avl}\langle\texttt{res},\texttt{s2},\texttt{h2}\rangle;\\ \quad\texttt{t1}\neq\texttt{null} \rightarrow \texttt{requires avl}\langle\texttt{t2},\texttt{s2},\texttt{h2}\rangle * \texttt{avl}\langle\texttt{t1},\texttt{s1},\_\rangle\\ \quad\quad\texttt{ensures avl}\langle\texttt{res},\texttt{s1}{+}\texttt{s2},\_\rangle\};\end{array}$$

However, let us note that there is a redundancy in this specification, namely the same predicate $\texttt{avl}\langle\texttt{t2},\texttt{s2},\texttt{h2}\rangle$ appears on both branches of the case construct. After the need for a case construct which was already discussed in Sec 1, this is the second deficiency we shall address in our specification mechanism, that is due to a lack of sharing in the logic formula which in turn causes repeated proving of identical sub-formulae. To provide for better sharing of the verification process, we propose to use *staged* formulae of the form $(\Phi_1 \textbf{ then } \Phi_2)$, to allow sub-formula Φ_1 to be proven prior to Φ_2.

Though $(\Phi_1 \textbf{ then } \Phi_2)$ is semantically equivalent to $(\Phi_1 * \Phi_2)$, we stress that the main purpose of adding this new structure is to support more effective verification with

the help of specifications with less redundancy. By itself, it is not meant to improve the expressivity of our specification, but rather its effectiveness. Nevertheless, when it is used in combination with the case construct, it could support case analysis of logical variables to ensure successful verification. The same structuring mechanisms can be used by formulae in both predicate definitions and pre/post specifications.

Getting back to the AVL merging example, the redundancy in the specification can be factored out by using a staged formulae, as follows:

$$\begin{array}{l}\texttt{requires avl}\langle \texttt{t2}, \texttt{s2}, \texttt{h2}\rangle \texttt{ then}\\ \texttt{case}\{\texttt{t1} = \texttt{null} \rightarrow \texttt{ensures avl}\langle \texttt{res}, \texttt{s2}, \texttt{h2}\rangle;\\ \qquad \texttt{t1}{\neq}\texttt{null} \rightarrow \texttt{requires avl}\langle \texttt{t1}, \texttt{s1}, \_\rangle \texttt{ ensures avl}\langle \texttt{res}, \texttt{s1}{+}\texttt{s2}, \_\rangle\};\end{array}$$

During the verification process, when reaching a call to the AVL merging method, the current program state must entail the method's precondition. Since the entailment process needs to explore both branches of the specification, the $\texttt{avl}\langle \texttt{t2}, \texttt{s2}, \texttt{h2}\rangle$ node will be proven twice for each method call. By using staged formulae, the second specification will force the common formula to be proved only once. Although the two specifications capture the same information, the second version requires much less proving effort. For this example, there was a 40% reduction in verification time by our system, due solely to the presence of staged formulae.

For the general case, if $\texttt{x}$ denotes the number of heap nodes/predicates that are shared in the consequent formula, and $\texttt{y}$ the number of possible matchings from the antecedent, then the number of redundant matchings that are eliminated is $(\texttt{x} - 1) * \texttt{y}$. An analogy can be made between the use of the staged formula and the use of the binary decision diagram (BDD) as an intermediate representation for SAT formulae to support better sharing of identical sub-formulae [4]. Where applicable, we expect staged formulae to improve the effectiveness of verification.

2.2 Example 2

Parameter instantiation is needed primarily for connecting the logical variables between precondition and postcondition of specifications. Traditionally, manual instantiation of ghost variables has played this role. In this paper, we propose two new mechanisms, early and late instantiations, to support automatic instantiations of logical variables. As an example, consider a data node `cell` and a predicate `cellPred` defined as follows:

$$\begin{array}{l}\texttt{data cell } \{\texttt{ int val}\}\\ \texttt{cellPred}\langle \texttt{root}, \texttt{i}\rangle \equiv \texttt{root}{=}\texttt{null} \wedge \texttt{i}{\leq}3 \ \vee \ \texttt{root} \mapsto \texttt{cell}\langle \_\rangle \wedge \texttt{i}{>}3\end{array}$$

To highlight the difference between early and late instantiations, we shall consider two separate proof obligations. The first one is given below.

$$\texttt{p} \mapsto \texttt{cell}\langle \_\rangle \vdash (\texttt{cellPred}\langle \texttt{p}, \texttt{j}\rangle {\wedge} \texttt{j}{>}2) {*} \Phi_{\texttt{r}}$$

At this point, we first need to match a heap predicate $\texttt{cellPred}\langle \texttt{p}, \texttt{j}\rangle$ on the RHS with a data node $\texttt{p} \mapsto \texttt{cell}\langle \_\rangle$ on the LHS to obtain an instantiation for the variable $\texttt{j}$. A fundamental question is whether the variable instantiation could occur for just the predicate $\texttt{cellPred}\langle \texttt{p}, \texttt{j}\rangle$ (we refer to this as *early instantiation*), or it has to be for the entire formula $\texttt{cellPred}\langle \texttt{p}, \texttt{j}\rangle \wedge \texttt{j}{>}2$ (known as *late instantiation*). By default, our

system uses early (or implicit) instantiation for variables that are not explicitly declared. In this scenario, early instantiation $\mathtt{j}{>}3$ is obtained when folding with the predicate $\mathtt{cellPred}\langle \mathtt{p}, \mathtt{j}\rangle$. This instantiation is transferred to the LHS. Consequently, we obtain a successful proof below.

$$\mathtt{j}{>}3 \vdash (\mathtt{j}{>}2){*}\Phi_r$$

Now, let us consider a second proof obligation that will require late instantiation:

$$\mathtt{p}{=}\mathtt{null} \vdash (\mathtt{cellPred}\langle \mathtt{p}, \mathtt{j}\rangle{\wedge}\mathtt{j}{>}2){*}\Phi_r$$

Similar to the previous case, we will first use a default early instantiation mechanism. After matching $\mathtt{cellPred}\langle \mathtt{p}, \mathtt{j}\rangle$, we obtain the instantiation $\mathtt{j}{\leq}3$. However, moving only this binding to the LHS is not enough, causing the proof below to fail.

$$\mathtt{p}{=}\mathtt{null} \wedge \mathtt{j}{\leq}3 \vdash (\mathtt{j}{>}2){*}\Phi_r$$

To support late instantiation for variable $\mathtt{j}$, we declare it explicitly using $[\mathtt{j}]$ below:

$$\mathtt{p}{=}\mathtt{null} \vdash ([\mathtt{j}]\ \mathtt{cellPred}\langle \mathtt{p}, \mathtt{j}\rangle{\wedge}\mathtt{j}{>}2){*}\Phi_r$$

This time variable $\mathtt{j}$ is kept on the RHS until the end of the entailment. As its proof below succeeds, the instantiation for $\mathtt{j}$ will be captured in the residue as $\Phi_r{=}\mathtt{j}{\leq}3{\wedge}\mathtt{j}{>}2$.

$$\mathtt{p}{=}\mathtt{null} \vdash (\exists \mathtt{j}.\mathtt{j}{\leq}3{\wedge}\mathtt{j}{>}2){*}\Phi_r$$

Though late instantiation is more general, it may require existential quantifications over a larger formula. Hence, by default, we prefer to use early instantiation where possible, and leave it to the user to manually declare where late instantiation is mandated.

3 Structured Specifications

Pre/Post.	Z	$::= \exists v_1^*{\cdot}\mathsf{Y}_1 \ldots \exists v_n^*{\cdot}\mathsf{Y}_n$	multiple specs
	Y	$::= \mathtt{case}\{\pi_1{\Rightarrow}\mathsf{Z}_1; \ldots; \pi_n{\Rightarrow}\mathsf{Z}_n\}$	case construct
		$\mid\ \mathtt{requires}\ [w^*]\ \Phi\ [\mathtt{then}]\ \mathsf{Z}$	staged spec
		$\mid\ \mathtt{ensures}\ \mathsf{Q}$	post
Formula	Q	$::= \bigvee \exists v^*{\cdot}\mathsf{R}$	multiple disjuncts
	R	$::= \mathtt{case}\{\pi_1{\Rightarrow}\mathsf{Q}_1; \ldots; \pi_n{\Rightarrow}\mathsf{Q}_n\}$	case construct
		$\mid\ [w^*]\ \Phi\ [\mathbf{then}\ \mathsf{Q}]$	staged formula
	Φ	$::= \bigvee \exists v^* \cdot (\kappa \wedge \pi)$	
Heap formula	κ	$::= \mathbf{emp} \mid v \mapsto \mathsf{c}\langle v^*\rangle \mid p\langle v^*\rangle \mid \kappa_1 * \kappa_2$	
Pure formula	π	$::= \ldots$	

Fig. 1. Syntax for Structured Specifications

We shall now focus on the structured specifications mechanism. Fig 1 provides a syntactic description where Z denotes structured (pre/post) specifications, while Q denotes structured formulae that may be used for pre/post specifications, as well as for predicate definitions. Apart from multiple specifications, our new syntax includes case constructs and staged formulae.

For structured specification, the `requires` keyword introduces a part of precondition through a staged specification. The postcondition is captured after each `ensures` keyword, which must appear as a terminating branch for the tree-like specification format. We support late instantiation via variables w^*, from `requires` $[w^*]\ \Phi\ \mathsf{Z}$ and $[w^*]\ \Phi\ [\mathbf{then}\ \mathsf{Q}]$ at the end of proving Φ. To minimise user annotations, our system automatically determines the other unbound variables (different from those to be late instantiated) as either existential or to be early instantiated.

Our construct to support case analysis is $\texttt{case}\{\pi_1 \Rightarrow \mathsf{Z}_1; \ldots; \pi_n \Rightarrow \mathsf{Z}_n\}$ for specification, and $\texttt{case}\{\pi_1 \Rightarrow \mathsf{Q}_1; \ldots; \pi_n \Rightarrow \mathsf{Q}_n\}$ for formula. We impose the following three conditions on $\pi_1, \ldots, \pi_n$:

(i) are *restricted* to only pure constraints, without any heap formula.
(ii) are *exclusive*, meaning that $\forall i, j \cdot i \neq j \rightarrow \pi_i \wedge \pi_j = \texttt{false}$.
(iii) are *exhaustive*, meaning that $\pi_1 \vee \ldots \vee \pi_n = \texttt{true}$.

Condition (i) is imposed since pure formula can be freely duplicated. Condition (ii) is imposed to avoid conjunction over the heap-based formula. If absent, each heap state may have to satisfy multiple case branches. Condition (iii) is needed for soundness of case analysis which requires all scenarios to be considered. To illustrate, consider:

$$[(\texttt{w} : \texttt{t})^*]\ \Phi\ \texttt{case}\{\texttt{x}=\texttt{null} \Rightarrow \mathsf{Q}_1;\ \texttt{x} \neq \texttt{null} \Rightarrow \mathsf{Q}_2\}$$

The first condition holds as the two guards, $\texttt{x}=\texttt{null}$ and $\texttt{x} \neq \texttt{null}$, are pure. Furthermore, our system checks successfully that the guards are exclusive $((\texttt{x}=\texttt{null} \wedge \texttt{x} \neq \texttt{null}) = \texttt{false})$ and exhaustive $((\texttt{x}=\texttt{null} \vee \texttt{x} \neq \texttt{null}) = \texttt{true})$.

3.1 Semantic Model for Structured Formulae

The semantics of our structured formula is similar to those given for separation logic [21], with extensions for the new structured formulae.

To define the model we assume sets *Loc* of locations (positive integer values), *Val* of primitive values, with $0 \in \mathit{Val}$ denoting `null`, *Var* of variables (program and logical variables), and *ObjVal* of object values stored in the heap, with $c[f_1 \mapsto \nu_1, .., f_n \mapsto \nu_n]$ denoting an object value of data type c where $\nu_1, .., \nu_n$ are current values of the corresponding fields $f_1, .., f_n$. Let $s, h \models \mathsf{Q}$ in Fig 2 denote the model relation, i.e. the stack s and heap h satisfy the constraint Q, with h, s from the following concrete domains:

$$\begin{array}{rcl} h \in \mathit{Heaps} & =_{df} & \mathit{Loc} \rightharpoonup_{fin} \mathit{ObjVal} \\ s \in \mathit{Stacks} & =_{df} & \mathit{Var} \rightarrow \mathit{Val} \cup \mathit{Loc} \end{array}$$

Note that each heap h is a finite partial mapping while each stack s is a total mapping, as in the classical separation logic [21, 9]. Function $\mathit{dom}(f)$ returns the domain of function f. The model relation for separation heap formulas is defined below. The model relation for pure formula $s \models \pi$ denotes that the formula π evaluates to `true` in s. Note that $h_1 \perp h_2$ indicates h_1 and h_2 are domain-disjoint, $h_1 \cdot h_2$ denotes the union of disjoint heaps h_1 and h_2. For the case of a data node, $v \mapsto \texttt{c}\langle v^* \rangle$, h has to be a singleton heap. On the other hand, a shape predicate defined by $p\langle v_{1..n} \rangle \equiv \mathsf{Q}$ may be inductively defined.

With the semantics of the structured formulae in place, we can provide a translation from a structured formula to its equivalent unstructured formula. This translation is formalised with $\mathsf{Q} \rightsquigarrow_T \Phi$, as shown below:

$$
\begin{array}{ll}
s,h \models \mathsf{Q} & \texttt{iff}\ \ \mathsf{Q} = \bigvee_{i=1}^{n} \exists v^* \cdot \mathsf{R}_i \text{ and } s,h \models \bigvee_{i=1}^{n} \exists v^* \cdot \mathsf{R}_i \\
s,h \models \bigvee_{i=1}^{n} \exists v_{i1..im} \cdot \mathsf{R}_i & \texttt{iff}\ \ \exists k \in \{1,..,n\} \cdot \exists \alpha_{k1..km} \cdot \\
 & \quad s[v_{k1} \mapsto \alpha_{k1}, .., v_{km} \mapsto \alpha_{km}], h \models \mathsf{R}_k \\
s,h \models [w_{i=1}^{n}] \Phi \ \texttt{then}\ \mathsf{Q} & \texttt{iff}\ \ \exists h_1, h_2 \cdot h_1 \perp h_2 \text{ and } h = h_1 \cdot h_2 \\
 & \quad \text{and } \exists \alpha_{1..n} \cdot s[w_1 \mapsto \alpha_1, .., w_n \mapsto \alpha_n], h_1 \models \Phi \text{ and } s, h_2 \models \mathsf{Q} \\
s,h \models \texttt{case}\{(\pi_i \Rightarrow \mathsf{Q}_i)_{i=1}^{n}\} & \texttt{iff}\ \ \forall k \in \{1,..,n\} \cdot (s,h \models \pi_k \ \rightarrow\ s,h \models \mathsf{Q}_k) \\
s,h \models \Phi_1 \vee \Phi_2 & \texttt{iff}\ \ s,h \models \Phi_1 \text{ or } s,h \models \Phi_2 \\
s,h \models \exists v_{1..n} \cdot \kappa \wedge \pi & \texttt{iff}\ \ \exists \alpha_{1..n} \cdot s[v_1 \mapsto \alpha_1, .., v_n \mapsto \alpha_n], h \models \kappa \\
 & \quad \text{and } s[v_1 \mapsto \alpha_1, .., v_n \mapsto \alpha_n] \models \pi \\
s,h \models \kappa_1 * \kappa_2 & \texttt{iff}\ \ \exists h_1, h_2 \cdot h_1 \perp h_2 \text{ and } h = h_1 \cdot h_2 \\
 & \quad \text{and } s,h_1 \models \kappa_1 \text{ and } s,h_2 \models \kappa_2 \\
s,h \models \texttt{emp} & \texttt{iff}\ \ \mathit{dom}(h) = \emptyset \\
s,h \models p \mapsto \texttt{c}\langle v^* \rangle & \texttt{iff}\ \ \textit{exists a data type decl. } \texttt{data c } \{t_1\ f_1, .., t_n\ f_n\} \\
 & \quad \text{and } h = [s(p) \mapsto r] \text{ and } r = \texttt{c}[f_1 \mapsto s(v_1), .., f_n \mapsto s(v_n)] \\
s,h \models p\langle v_{1..n} \rangle & \texttt{iff}\ \ \textit{exists a pred. def. } p\langle v_{1..n} \rangle \equiv \mathsf{Q} \text{ and } s,h \models \mathsf{Q}
\end{array}
$$

Fig. 2. Model for Structured Formulae

$$
\frac{\forall i \cdot \mathsf{Q}_i \rightsquigarrow_T \Phi_i}{\texttt{case}\{\pi_i \Rightarrow \mathsf{Q}_i\}^* \rightsquigarrow_T \bigvee (\Phi_i \wedge \pi_i)} \qquad \frac{\mathsf{Q} \rightsquigarrow_T \Phi}{[w^*]\ \Phi_1\ \texttt{then}\ \mathsf{Q} \rightsquigarrow_T \Phi_1 * \Phi}
$$

$$
\frac{\forall i \cdot \mathsf{R}_i \rightsquigarrow_T \Phi_i}{\bigvee \exists v^* \cdot \mathsf{R}_i \rightsquigarrow_T \bigvee \exists v^* \cdot \Phi_i} \qquad \frac{}{[w^*]\ \Phi \rightsquigarrow_T \Phi}
$$

We make use of the semantics for structured formulae Q and for unstructured formula $\mathtt{\Phi}$ to prove the correctness of the given translation rules.

Theorem 3.1 (Correctness of Translation) *Given Q and Φ such that $\mathsf{Q} \rightsquigarrow_T \Phi$: for all s,h, $s,h \models \mathsf{Q}$ if and only if $s,h \models \Phi$.*

Proof: By structural induction on Q.

4 Modular Verification

The main goal of structured specification is to support a modular verification process that could be carried out efficiently and precisely. In this section, we propose a set of rules to help generate Hoare-style triples for code verification, together with entailment checking to support proof obligations over the structured formulae domain.

4.1 Building Verification Rules

Program verification is typically formalised using Hoare triples of the form $\{pre\}\mathtt{e}\{post\}$, where *pre* and *post* are the initial and final states of the program code ($\mathtt{e}$) in some logic. Our verification system uses separation logic, where a Hoare-style specification $\{pre\}\mathtt{e}\{post\}$ is valid, denoted as $\models \{pre\}\mathtt{e}\{post\}$, if and only if, for all states (s,h) that $s,h \models pre$, if the execution of $\mathtt{e}$ starting from (s,h) does not lead to memory errors and terminates in a state (s_1,h_1), then $s_1,h_1 \models post$.

To better support structured specifications and case analysis, we propose a new triple of the form $\{\!|\Phi|\!\}\ e\ \{\!|\mathsf{Z}|\!\}$, with *pre* being an unstructured formula and Z being the structured specification. We use structured specifications in the poststate because our case

analysis is guided from the post-states. In contrast, unstructured formulae are used in the prestate since the structured form is unnecessary here. The semantic meaning of this new triple is defined as follows:

Definition 4.1 *The validity of* $\{\!|\Phi|\!\}\ \mathtt{e}\ \{\!|Z|\!\}$ *is defined inductively over the structure of* Z*. That is:*

- *if* $Z \equiv \mathtt{ensures}\ Q$: $\models \{\!|\Phi|\!\}\ \mathtt{e}\ \{\!|Z|\!\} \iff \models \{\Phi\}\mathtt{e}\{Q\}$;
- *if* $Z \equiv \mathtt{requires}\ \Phi_1\ [\mathtt{then}]\ Z_1$: $\models \{\!|\Phi|\!\}\ \mathtt{e}\ \{\!|Z|\!\} \iff \models \{\!|\Phi * \Phi_1|\!\}\ \mathtt{e}\ \{\!|Z_1|\!\}$;
- *if* $Z \equiv \mathtt{case}\{\pi_1 \Rightarrow Z_1; \ldots; \pi_n \Rightarrow Z_n\}$: $\models \{\!|\Phi|\!\}\ \mathtt{e}\ \{\!|Z|\!\} \iff$
  $\iff \forall i \in \{1,..,n\} \cdot \models \{\!|\Phi \wedge \pi_i|\!\}\ \mathtt{e}\ \{\!|Z_i|\!\}$;
- *if* $Z \equiv (\exists v_1^* \cdot Y_1 \ldots \exists v_n^* \cdot Y_n)$: $\models \{\!|\Phi|\!\}\ \mathtt{e}\ \{\!|Z|\!\} \iff$
  $\iff \forall i \in \{1,..,n\} \cdot \models \{\!|\Phi|\!\}\ \mathtt{e}\ \{\!|\exists v_i^* \cdot Y_i|\!\})\ \square$

$$\frac{H=[(v{:}t)^*, (u{:}t)^*] \quad G = \mathit{prime}(H)+H+[\mathtt{res}{:}t_0] \quad G \vdash \{\!|\bigwedge (v'{=}v)^* \wedge \bigwedge (u'{=}u)^*|\!\}\ \mathtt{code}\ \{\!|Z|\!\}}{\vdash t_0\ mn\ ((t\ v)^*,\ (\mathit{ref}\ t\ u)^*)\ Z\ \{\ \mathtt{code}\ \}}\ [\textsc{fv-meth}]$$

$$\frac{\mathit{fresh}\ \ nv^* \quad \rho=[(v \rightarrow nv)^*] \quad \forall i \cdot G \vdash \{\!|\Phi|\!\}\ \mathtt{code}\ \{\!|\rho Y_i|\!\}}{G \vdash \{\!|\Phi|\!\}\ \mathtt{code}\ \{\!|\exists v_1^* \cdot Y_1 .. \exists v_n^* \cdot Y_n|\!\}}\ [\textsc{fv-multi-specs}]$$

$$\frac{\{w*\} \cap \mathit{Vars}(G) = \{\} \quad G_1 = G + [(w:t)^*] \quad G_1 \vdash \{\!|\Phi_1 * \Phi_2|\!\}\ \mathtt{code}\ \{\!|Z|\!\}}{G \vdash \{\!|\Phi_1|\!\}\ \mathtt{code}\ \{\!|\mathtt{requires}\ [(w:t)^*]\ \Phi_2\ Z|\!\}}\ [\textsc{fv-requires}]$$

$$\frac{V = \mathit{PassByValue}(G) \quad \vdash \{\Phi\}\ \mathtt{code}\ \{\Phi_2\} \quad \exists \mathit{prime}(V) \cdot \Phi_2 \vdash^{\mathtt{emp}}_{\{\}} Q * S \quad S \neq \{\}}{G \vdash \{\!|\Phi|\!\}\ \mathtt{code}\ \{\!|\mathtt{ensures}\ Q|\!\}}\ [\textsc{fv-ensures}]$$

$$\frac{\forall i \in \{1,..,n\} \cdot G \vdash \{\!|\Phi \wedge \pi_i|\!\}\ \mathtt{code}\ \{\!|Z_i|\!\}}{G \vdash \{\!|\Phi|\!\}\ \mathtt{code}\ \{\!|\mathtt{case}\{\pi_1 \Rightarrow Z_1; \ldots; \pi_n \Rightarrow Z_n\}|\!\}}\ [\textsc{fv-case}]$$

Fig. 3. Building Verification Rules for Structured Specifications

Our main verification rules are given in Fig. 3. Note that G records a list of variables (including `res` as result of the code) visible to the code verifier. Our specification formulae use both primed and unprimed notations, where primed notations represent the latest values of program variables, and unprimed notations denote either logical variables or initial values of program variables.

The verification of method declarations is described by the $[\textsc{fv-meth}]$ rule. It verifies the method body `code` against the specification Z, as indicated by the rule. The function $\mathit{prime}(\{v_1,..,v_m\})$ returns the primed version $\{v_1',..,v_m'\}$. The third line of the premise deals with the verification task $G \vdash \{\!|\bigwedge (v'{=}v)^* \wedge \bigwedge (u'{=}u)^*|\!\}\ \mathtt{code}\ \{\!|Z|\!\}$, where the precondition indicates that the latest values of program variables are the same as their initial values. The other rules are syntax-directed and rely on the structure of the specification Z.

The rule $[\textsc{fv-multi-specs}]$ deals with the case where the post-state is a multi-specification. It verifies the code against each of the specifications. Note that the substitution ρ replaces variables v^* with fresh variables nv^*. The rule $[\textsc{fv-requires}]$ deals with the case where the post-state starts with a `requires` clause. In this case, the formula in the `requires` clause is added to the pre-state (by separation conjunction) before

verifying the code against the remaining part of the specification in the post-state. The variables for late instantiation (w^*) are also attached to the end of the list G. The rule [FV–ENSURES] deals with the case where the post-state starts with an `ensures` clause. It invokes our forward verification rules to derive the strongest postcondition Φ_2 for the normal Hoare triple $\{\Phi\}\texttt{code}\{\Phi_2\}$ and invokes the entailment prover (described in the next section) to check that the derived post-state Φ_2 subsumes the given post-condition Q (The test $S \neq \{\}$ signifies the success of this entailment proof). Note that V denotes the set of pass-by-value parameters that are not modified by the procedure. Hence, their values (denoted by primed variables) are ignored in the postcondition, even if the program code may have updated these parameters. The last rule [FV–CASE] deals with the case where the post-state is a case specification. It verifies in each case the specification Z_i is met when the guard π_i is assumed in the pre-state.

To illustrate the generation of the verification tasks, consider the AVL merging given in Section 2.1. By applying the rules from Figure 3, two Hoare triples are produced.

$$\vdash \{\texttt{avl}\langle \texttt{t2}, \texttt{s2}, \texttt{h2}\rangle \wedge \texttt{t1}{=}\texttt{null}\}\ \texttt{code}\ \{\texttt{avl}\langle \texttt{res}, \texttt{s2}, \texttt{h2}\rangle\}$$
$$\vdash \{\texttt{avl}\langle \texttt{t1}, \texttt{s1}, \_\rangle * \texttt{avl}\langle \texttt{t2}, \texttt{s2}, \texttt{h2}\rangle \wedge \texttt{t1}{\neq}\texttt{null}\}\ \texttt{code}\ \{\texttt{avl}\langle \texttt{res}, \texttt{s1}{+}\texttt{s2}, \_\rangle\}$$

Theorem 4.1 (Soundness of Verification) *Our verification rules are* sound*. That is, given a program* `code`*, an unstructured formula Φ, and a structured specification Z, if our system derives a proof, $\vdash \{\!|\Phi|\!\}\ \texttt{code}\ \{\!|\mathsf{Z}|\!\}$, then we have $\models \{\!|\Phi|\!\}\ \texttt{code}\ \{\!|\mathsf{Z}|\!\}$.*

Proof: It follows from the soundness of our underlying verification system (i.e. the one without structured specifications) [17], the definition 4.1, and the soundness of the entailment prover enriched with structured formulae (described in the next section).

4.2 Entailment for Structured Formula

Given formulae Φ_1 and Q_2, our entailment prover checks if Φ_1 entails Q_2, that is if in all heaps satisfying Φ_1, we can find a subheap satisfying Q_2.

The main features of our entailment prover are that, besides determining if the entailment relation holds, it also infers the residual heap of the entailment, that is a formula Φ_R such that $\Phi_1 \vdash \mathsf{Q}_2 * \Phi_R$ and derives the predicate parameters. The relation is formalized using a judgment of the form $\Phi_1 \vdash^{\kappa}_{V} \mathsf{Q}_2 * \Phi_R$, which is a shorthand for $\Phi_1 * \kappa \vdash \exists V \cdot (\mathsf{Q}_2 * \kappa) * \Phi_R$. Note that κ denotes the consumed heap, while V is a set, $\{v^*, E{:}w^*\}$, containing the existential variables encountered, v^*, together with the variables w^* for late instantiation, .

To support proof search, we have also generalised the entailment checking procedure to return a set of residues S_R: $\Phi_1 \vdash^{\kappa}_{V} \mathsf{Q}_2 * S_R$.This entailment succeeds when S_R is non-empty, otherwise it is deemed to have failed. The multiple residual states captured in S_R signify different search outcomes during proving. Our entailment procedure relies on unfolding and folding of the predicate definitions. Unfolding refers to a single inlining of a predicate in the antecedent, while folding is a recursive entailment with the body of a predicate in the consequent. In the current paper, we enhance the entailment proving procedure to handle structured formulae in the consequent. The main rules are given in Figure 4. Take note that we make use of a method *mark*(V, w^*) , which marks

the variables to be late instantiated, w^*, by removing them from the existential variables stored in V and adding them as $E : w^*$:

$$\mathit{mark}(V, w^*) = (V - \{w^*\}) \cup \{(E : w)^*\}$$

The rule [ENT−FORMULA] makes use of the aforementioned marking method in order to mark the fact that variables w^* are to be late instantiated, whereas rule [ENT−EXIST] adds the existentially quantified variables v^* to the set V.

$$\frac{[\textsc{ent}-\textsc{formula}]\quad \Phi \vdash^{\kappa}_{\mathit{mark}(V,w^*)} (\Phi_1) * \mathsf{S}}{\Phi \vdash^{\kappa}_{V} [w^*]\ \Phi_1 * \mathsf{S}} \qquad \frac{[\textsc{ent}-\textsc{case}]\quad \forall i \cdot \Phi \wedge \pi_i \vdash^{\kappa}_{V} \mathsf{Q}_i * \mathsf{S}_i}{\Phi \vdash^{\kappa}_{V} \mathtt{case}\{\pi_i \Rightarrow \mathsf{Q}_i\}^* * (\bigvee \mathsf{S}_i)}$$

$$\frac{[\textsc{ent}-\textsc{ensures}]\quad \mathsf{Q} \rightsquigarrow_T \Phi_1}{\Phi \vdash^{\kappa}_{V} (\mathtt{ensures}\ \mathsf{Q}) * (\Phi * \Phi_1)} \qquad \frac{[\textsc{ent}-\textsc{staged}-\textsc{formula}]\quad \Phi \vdash^{\kappa}_{\mathit{mark}(V,w^*)} (\Phi_1) * \mathsf{S} \quad \mathsf{S} \vdash^{\kappa}_{V-\{w^*\}} (\mathsf{Q}) * \mathsf{S}_2}{\Phi \vdash^{\kappa}_{V} ([w^*]\ \Phi_1\ \mathtt{then}\ \mathsf{Q}) * \mathsf{S}_2}$$

$$\frac{[\textsc{ent}-\textsc{rhs}-\textsc{or}]\quad \forall i \cdot \Phi \vdash^{\kappa}_{V} \mathsf{R}_i * \mathsf{S}_i}{\Phi \vdash^{\kappa}_{V} \bigvee \mathsf{R}_i * (\bigcup \mathsf{S}_i)} \qquad \frac{[\textsc{ent}-\textsc{exist}]\quad \Phi \vdash^{\kappa}_{V \cup \{v^*\}} \mathsf{R} * \mathsf{S}}{\Phi \vdash^{\kappa}_{V} \exists v^* \cdot \mathsf{R} * \mathsf{S}}$$

Fig. 4. Entailment for Structured Formula

In the rule for staged formula, [ENT−STAGED−FORMULA], the instantiation for the variables w^* takes place in the first stage, Φ_1. As instantiation moves the corresponding bindings to the LHS (or antecedent of entailment), the variables w^* must be removed from the set of existentially quantified variables when entailing the rest of the formula, Q. At the end of the entailment proving, the variables that were marked as late-instantiated are existentially quantified in the residue state. The generalised entailment with a set of n formulae in the antecedent is an abbreviation of the n entailments, as illustrated below:

$$\frac{\forall i \in \{1, .., n\} \cdot \Phi_i \vdash^{\kappa}_{V} (\mathsf{Q}) * \mathsf{S}_i}{\{\Phi_1, .., \Phi_n\} \vdash^{\kappa}_{V} (\mathsf{Q}) * \bigcup_{i=1}^{n} \mathsf{S}_i}$$

The rule [ENT−CASE] adds the pure term π_i to the antecedent. This rule requires a lifted disjunction operation defined as $S_1 \vee S_2 \equiv \{\Phi_1 \vee \Phi_2 | \Phi_1 \in S_1, \Phi_2 \in S_2\}$ when applied to two sets of states, S_1, S_2.

While a successful entailment of one disjunct suffices for the entailment of a disjunctive formula, our entailment rule [ENT−RHS−OR] facilitates a proof search by trying to entail each of the RHS disjuncts separately. Therefore, the residue state must contain the union of all residues corresponding to the proof search from a set of entailments, $\forall i \cdot \Phi \vdash^{\kappa}_{V} \mathsf{R}_i * \mathsf{S}_i$.

Take note that, at each call site, the forward verification procedure ensures that the method's precondition is satisfied and assumes the method's postcondition. This is achieved by entailing a formula denoting a specification of the Z form. As the corresponding entailment rules are similar to those for the entailment of a structured formula given in Figure 4, we omit them for brevity. The only unusual rule is [ENT−ENSURES]

that is needed when entailing the actual postcondition `ensures Q`. In this case, the postcondition is added to the residual state in unstructured form, immediately after the translation $\mathsf{Q} \rightsquigarrow_T \Phi_1$ to unstructured form.

Theorem 4.2 (Soundness of Entailment) *Given Φ, Q such that $s, h \models \Phi$, if $\Phi \vdash^{\kappa}_{V} \mathsf{Q} * \Phi_r$ for some Φ_r, then $s, h \models \mathsf{Q} * \Phi_r$. That is, for all program states in which Φ holds if $\Phi \vdash^{\kappa}_{V} \mathsf{Q} * \Phi_r$ then $\mathsf{Q} * \Phi_r$ holds.*

Proof: By structural induction on Q.

5 Experiments

We have built a prototype system using Objective Caml. The proof obligations generated by our verification are discharged using some off-the-shelf constraint solvers (like Omega Calculator [20]) or theorem provers (like MONA [13]). The specification mechanism works with any constraint domain, as long as a corresponding prover for the domain is available. The specific domains that our verifier currently supports, includes linear (Omega Calculator, Z3, CVC-lite) and non-linear arithmetic (Redlog), set (MONA, Isabelle bag tactic) and list properties (a Coq tactic). Though the current paper highlighted mostly simpler specifications, our benchmark included the verification of functional correctness properties, such as sortedness and permutation.

We have conducted preliminary experiments by testing our system on a suite of examples summarized in Figure 5. These examples are small but can handle data structures with sophisticated shape and size properties such as sorted lists, balanced trees, etc., in a uniform way. Methods "insert" and "delete" refer to the insertion and deletion of a value into/from the corresponding data structure, respectively. Method "del_first" deletes the node at the head in a circular list. Moreover, we verify a suite of sorting algorithms, which receive as input an unsorted singly-linked list and return a sorted list. Verification time for each function includes the time to verify all functions that it calls. We compare the timings obtained with and without case analysis.

Take note that for each of the verified methods, in order to compare the results obtained with and without case analysis, we provided specifications with the same level of modularity through specifications with multiple pre/post. FAIL for the "without case" means it did not verify functional correctness (including memory safety). This is due the absence of case analysis that would have been provided by the missing case spec.

Preliminary results indicate that case analysis improves both the completeness and the performance of our system. From the completeness point of view, case analysis is important for verifying a number of examples that would *fail* otherwise. For instance, the method implementing the selection sort algorithm over a linked list fails when it is written with multiple specification instead of the case construct. The same scenario is encountered for the method inserting/deleting a node of red black tree, and for the method appending two list segments. The case construct thus helps our system to verify more examples successfully. Regarding the performance, the timings obtained when using case analysis are smaller, taking on average 21% less computation time than those obtained without case analysis. The improvements are due to earlier pruning of `false` contexts with the help of case constructs and optimizations of the case entailment rule.

Program Codes	LOC	Timings (in seconds)		speed gain (%)
		with case	*without* case	
Linked List		verifies length		
delete	20	0.65	0.89	26
append	14	0.30	0.39	23
List Segment		verifies length		
append	11	0.95	*failed*	-
Circular Linked List		verifies length + circularity		
del_first	15	0.35	0.41	15
insert	10	0.28	0.35	20
Doubly Linked List		verifies length + double links		
insert	18	0.35	0.52	33
delete	29	0.94	1.27	26
Sorted List		verifies bounds + sortedness		
insert	17	0.71	0.96	26
delete	21	0.60	0.68	22
insertion_sort	45	0.92	1.35	32
selection_sort	52	1.24	*failed*	-
bubble_sort	42	1.95	2.92	43
merge_sort	105	2.01	2.53	31
quick_sort	85	1.82	2.47	26
AVL Tree		verifies size + height + balanced		
insert	169	32.27	39.48	19
delete	287	85.1	97.30	13
Perfect Tree		verifies height + perfectness		
insert	89	0.73	0.99	26
Red-Black Tree		verifies size + black-height		
insert	167	5.44	*failed*	-
delete	430	22.43	*failed*	-

Fig. 5. Verification Times for Case Construct vs Multiple Pre/Post

We also investigated the performance gain that can be attributed to the use of staged formulae. We observed that the timings improved on average by 20%. Noteworthy examples include the AVL insertion (from 32.27s to 22.93s) and AVL deletion (from 85.1s to 81.6s).

We may conclude from our experiments that structured specifications together with case analysis give better precision to our verification system while also improving its performance, when compared to corresponding unstructured specifications.

6 Related Work and Conclusion

Previous works on enhancing pre/post specifications [14, 12] were mainly concerned with improving modularity to allow easier understanding of specifications. With this objective, multiple specifications and redundant representations were advocated as the primary machinery. In the context of shape analysis, Chang and Rival [6] make use of `if` notation for defining inductive checkers. However, the conditional gets approximated to disjunction during the actual analysis. Verification wise, the three structured specification mechanisms that we have proposed are not available in existing tools, such as JML [5], Spec# [1], Dafny [15], JStar [8] and VeriFast [10]. The closest relationships may be summarized, as follows. JML supports specification cases, in the

form of multiple pre/post conditions, for better modularity and clarity of specifications. Our case constructs also intend to provide better guidance to the verification process. Spec#/Dafny supports ghost variables for manual instantiation (by user) of logical variables. In contrast, our early/late instantiation mechanisms provided two solutions to automatic instantiation of logical variables. Overall, little attempt has been made to add specification structures that can help produce a better verification outcome.

On timings, we did not compare with Spec# and Dafny, since our benchmark on heap-manipulating programs is not properly covered by their specification logic. Regarding JStar, it currently uses logics involving only shapes and equalities, it does not support more expressive properties, like set and numeric properties, needed by our benchmark. Lastly, VeriFast requires more user intervention in the form of explicit unfolding and folding of the abstract predicates through ghost statements.

In a distributed systems setting, Seino et al [22] present a case analysis meant to improve the efficiency of protocol verification, which involves finding appropriate predicates and splitting a case into multiple sub-cases based on the predicates. In order to cover all the possible case splits, they use a special type of matrix. Pientka [19] argues for the need of case analysis in inductive proofs. The potential case splits are selected heuristically, based on the pattern of the theorem. A case split mechanism has been used by Brock et al [3] to guide case analysis during proving. Jhala and McMillan [11] used a temporal case splitting in order to specialize the properties to be proven, so that they depend on only a finite part of the overall state. As opposed to the previous works, our current proposal is to incorporate structured mechanisms within the specification mechanism itself for guiding the case analysis, existential instantiation or staged proving.

Some existing theorem provers use tactics as a way to automate or semi-automate proofs, and our system can take advantage of them through lower-level pure proofs. However, for Hoare-style specification and verification, we have chosen to design a structured specification (rather than another tactic language) for the following reasons:

- It can be provided at a higher-level that users can understand more easily, since it is closer to specification mechanism rather than the (harder) verification process.
- It is more portable, as specification are tied to program codes, while tactic language tend to be prover-specific requiring the invoked prover to understand the relevant commands. Our approach basically breaks down larger (hard) proofs into smaller (simpler) proofs that any prover could more easily and more effectively handle, as confirmed by our experiments.
- Specification can be transformed (or restructured) which allows us to heuristically infer structured specifications from unstructured counterparts. A version of this translation from unstructured formula to structured formula has been implemented in our system. Though this can never be as good as that provided by expert users, it can nevertheless be used to handle most of the straightforward cases for legacy specifications, leaving the harder unverified examples to be handled by users.

The current paper has pioneered a novel approach towards resolving two key problems of verification, namely better modularity and better completeness through a new form of structured specification. Our proposal has been formalized and implemented with a promising set of experimental results.

Acknowledgement We thank the anonymous reviewers for their insightful feedback on this work. The work was supported by NUS Grant R-252-000-366-112, MoE Grant R-252-000-444-112 and EPSRC Grant EP/G042322.

References

1. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS*, volume 3362, pages 49–69. Springer-Verlag, LNCS, 2004.
2. J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, Springer LNCS 4111, pages 115–137, 2006.
3. B. Brock, M. Kaufmann, and J. Strother Moore. ACL2 Theorems About Commercial Microprocessors. In *FMCAD*, pages 275–293, 1996.
4. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35:677–691, 1986.
5. L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *Software Tools for Technology Transfer*, 2005.
6. B.-Y. E. Chang and X. Rival. Relational inductive shape analysis. In *POPL*, pages 247–260, 2008.
7. W.-N. Chin, C. David, H. H. Nguyen, and S. Qin. Multiple pre/post specifications for heap-manipulating methods. In *HASE*, pages 357–364, 2007.
8. D. Distefano and M. J. Parkinson. jStar: Towards Practical Verification for Java. In *OOPSLA*, 2008.
9. S. Ishtiaq and P.W. O'Hearn. BI as an assertion language for mutable data structures. In *ACM POPL*, pages 14–26, London, January 2001.
10. B. Jacobs, J. Smans, and F. Piessens. A Quick Tour of the VeriFast Program Verifier. In *APLAS*, pages 304–311, 2010.
11. R. Jhala and K. L. McMillan. Microarchitecture verification by compositional model checking. In *CAV*, pages 396–410, 2001.
12. H. B. M. Jonkers. Upgrading the pre- and postcondition technique. In *VDM*, pages 428–456, London, UK, 1991. Springer-Verlag.
13. N. Klarlund and A. Moller. MONA Version 1.4 - User Manual. BRICS Notes Series, January 2001.
14. G. T. Leavens and A. L. Baker. Enhancing the Pre- and Postcondition Technique for More Expressive Specifications. In *FM*, September 1999.
15. K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR (Dakar)*, pages 348–370, 2010.
16. M.J.Parkinson and G.M.Bierman. Separation logic and abstraction. In *ACM POPL*, pages 247–258, 2005.
17. H.H. Nguyen, C. David, S.C. Qin, and W.N. Chin. Automated Verification of Shape And Size Properties via Separation Logic. In *VMCAI*, Nice, France, January 2007.
18. P.W. O'Hearn, H. Yang, and J.C. Reynolds. Separation and Information Hiding. In *ACM POPL*, Venice, Italy, January 2004.
19. B. Pientka. A heuristic for case analysis. Technical report, 1995.
20. W. Pugh. The Omega Test: A fast practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 8:102–114, 1992.
21. J. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *IEEE LICS*, pages 55–74, Copenhagen, Denmark, July 2002.
22. T. Seino, K. Ogato, and K. Futatsugi. Mechanically supporting case analysis for verification of distributed systems. *IJPCC*, 2005.
23. K. Zee, V. Kuncak, and M. C. Rinard. An integrated proof language for imperative programs. In *PLDI*, pages 338–351, New York, NY, USA, 2009. ACM.