KIV: overview and VerifyThis competition

Gidon Ernst, Jörg Pfähler, Gerhard Schellhorn, Dominik Haneberg, Wolfgang Reif

Angaben zur Veröffentlichung / Publication details:

Ernst, Gidon, Jörg Pfähler, Gerhard Schellhorn, Dominik Haneberg, and Wolfgang Reif. 2015. "KIV: overview and VerifyThis competition." International Journal on Software Tools for Technology Transfer 17 (6): 677–94. https://doi.org/10.1007/s10009-014-0308-3.

KIV: overview and VerifyThis competition

Gidon Ernst · Jörg Pfähler · Gerhard Schellhorn · Dominik Haneberg · Wolfgang Reif

Abstract Members of our research group participated in the VerifyThis competition at FM 2012 in Paris using the interactive specification and verification system KIV. In this article we describe the KIV verification system and its latest additions. We discuss our solutions to the three VerifyThis problems and which features of KIV were used in solving them. We also report on our findings from performing the proofs.

Keywords Interactive theorem proving · Verification challenge · KIV · Separation logic

1 Introduction

The VerifyThis 2012 competition[1] consisted of three algorithms from different areas and the challenge was to specify them and to verify their correctness. Our group used KIV [23] to solve the three verification problems. KIV is an interactive specification and verification system that we are developing since the mid-1980s.

[1] http://fm2012.verifythis.org/.

G. Ernst · J. Pfähler (✉) · G. Schellhorn · D. Haneberg · W. Reif
Institute for Software and Systems Engineering,
Augsburg University, Augsburg, Germany
e-mail: joerg.pfaehler@informatik.uni-augsburg.de

G. Ernst
e-mail: ernst@informatik.uni-augsburg.de

G. Schellhorn
e-mail: schellhorn@informatik.uni-augsburg.de

D. Haneberg
e-mail: haneberg@informatik.uni-augsburg.de

W. Reif
e-mail: reif@informatik.uni-augsburg.de

The KIV system is a tool for software verification that supports different styles of modeling systems. At the core of KIV are a calculus for higher-order logic (i.e., simply-typed lambda calculus) and structured algebraic specifications.

KIV combines algebraic specifications for data types with different formalisms for describing system behavior. It uses a weakest precondition (*wp*-) calculus for abstract imperative programs [14] and sequential Java [31]. A standard specification style for software systems is Abstract State Machines (ASMs [3, 11]), where a collection of abstract programs specifies the operations of the system. To verify concurrent systems, KIV supports a temporal logic for programs.

KIV's strengths are a graphical user interface [12] that facilitates easy interactive proof development, a powerful simplifier to support the proof engineer by automatically simplifying proof goals, and a library of basic data types with a large number of already proven theorems.

Section 2 describes the KIV system and its core features in more detail. Section 2.1 describes the basic logic of KIV. Section 2.2 presents KIV's approach for specifying sequential software systems. The approach for concurrent ones is described just briefly in Sect. 2.3 as it played no role in the competition. In Sect. 2.4 we describe how to prove properties with KIV. The graphical user interface is presented in Sect. 2.5 and Sect. 2.6 discusses KIV's correctness management. One of the latest additions to KIV is an embedding of Separation Logic described in Sect. 2.7 to facilitate reasoning about programs that use dynamically allocated memory.

Section 3 describes our solutions for the three problems of the VerifyThis competition. Suffix arrays and longest common substring are described in Sect. 3.1. Section 3.2 discusses the verification of prefix sum. The third problem, deleting the minimum of a binary search tree, is solved using Separation Logic in Sect. 3.3. Section 4 is dedicated to a comparison of our approach to some of the solutions of other teams.

2 KIV overview

2.1 HOL and algebraic data types

The logical foundation of KIV is a calculus for higher-order logic (HOL). Structured algebraic specifications are used to build a hierarchy of data type definitions. Freely generated as well as non-free data types can be used. Basic specifications with data type or function declarations can be combined to structured specifications with the usual structuring operations, e.g. union, renaming, and enrichment. To support generic data types, it is possible to specify parameterized data types, e.g. sets. Parameterized data types can be instantiated explicitly, e.g. set of natural numbers.

Non-free data types (e.g. sets, heaps, and bags) are specified by giving appropriate axioms, centered around a data type-specific axiom of extensionality (which guarantees that a proper quotient type is constructed). Freely generated data types are much simpler because their required axioms are fixed and can be generated automatically. KIV offers a special syntax for defining free data types which removes the necessity to write down the obvious axioms. Figure 1 shows the KIV-specification for lists, an example of such a freely generated data type.

The signature generated from this data type consists of

- a new sort (list)
- the two constructor functions [] and + (written infix)
- two selectors *.first* and *.rest* (written postfix)
- variables x, y, z of sort list
- a function # counting the number of elements in a list
- a well-founded ordering ($<$), representing a subterm relation on the constructor terms.

Furthermore, KIV generates axioms stating that the constructor functions are injective, that different constructor functions yield different lists, and definitions for the selectors for non-empty lists ([].*first* and [].*rest* are unspecified). The system also generates structurally recursive definitions for the size

```
data specification
using elem;

list =    []
       | . + . (. .first : elem, . .rest : list);

variables x,y,z : list;
size functions   # .   : list → nat;
order predicates . < . : list × list;

end data specification
```

Fig. 1 Specification for lists

disjoint: $[] \neq a + x$;
injective: $a + x = b + y \leftrightarrow a = b \wedge x = y$;

first: $(a + x).\mathit{first} = a$;
rest: $(a + x).\mathit{rest} = x$;

size-base: $\#([]) = 0$;
size-rec: $\#(a + x) = 1 + \#(x)$;

less-base: $\neg(x < [])$;
less-rec: $x < b + y \leftrightarrow x = y \vee x < y$;

irreflexivity: $\neg(x < x)$;
transitivity: $x < y \wedge y < z \rightarrow x < z$;

Fig. 2 Generated axioms for lists

function and the order predicate. Figure 2 shows the generated axioms which are all implicitly universally quantified. The variables x, y, z come from the specification of lists, whereas a and b are elements (declared in the specification *elem*); this information is used by the type check. At the bottom of Fig. 2 there are also two axioms for irreflexivity and transitivity of the ordering. This is, strictly speaking, unnecessary as they could be proven, but they are generated nevertheless to avoid fthis proof effort. Lastly, KIV derives a proof rule for structural induction on lists from the specification. This induction principle follows from the property that every element of the sort is generated by a constructor term, which is an implicit property of every free data type.

To reduce the effort of specifying a system and to profit from earlier verification work done for common data structures, KIV comes with a large library of standard data types (natural numbers, integers, strings, lists, sets, bags, arrays, etc.) and associated theorems.

2.2 Specifying systems in KIV

On top of the algebraic approach KIV supports a *wp*-calculus for an abstract language of imperative programs as well as for sequential Java [31].

The syntax used for KIV's *wp*-calculus is Higher-order Dynamic Logic [14] which extends HOL by new formulae that contain programs: $[\alpha]\varphi$, $\langle\alpha\rangle\varphi$ and $\langle\!|\alpha|\!\rangle\varphi$, representing partial correctness, total correctness for deterministic programs, and total correctness for non-deterministic programs. For example, $\langle\alpha\rangle\varphi$, states that the program α started in the current state has a terminating run and φ will hold in the final state. KIV represents weakest-precondition statements with an appropriate DL-formula: $wp(\alpha, \varphi) = \langle\!|\alpha|\!\rangle\varphi$. Dynamic Logic can express Hoare-Triples as formulae: $\{\varphi\}\alpha\{\psi\}$ is equivalent to $\varphi \rightarrow [\alpha]\psi$. It also extends Hoare-Logic by allowing comparison between programs. For instance, program inclusion—when α terminates and yields a state in

```
procedures
factorial nat : nat;

axioms
⟨factorial(n;result)⟩ result = fac(n);

declaration
factorial(n; var result) {
  if n = 1 then {
    result := 1
  }
  else {
    factorial(n - 1; result);
    result := n * result
  }
}
```

Fig. 3 Procedure implementation and axiomatization

which φ holds so does program β—can be expressed as $\langle\alpha\rangle\varphi \rightarrow \langle\beta\rangle\varphi$. The programs in these new types of formulae are either in an abstract imperative language or in sequential Java. The abstract language supports the usual constructs (parallel assignment, if, while, mutually recursive procedures), where program variables and expressions range over the logical data types, i.e., algebraic operations can be used in expressions.

Procedures in KIV can either be implemented or only specified by axiomatizing a contract. Figure 3 shows the two ways of specifying a procedure. The first description of procedure `factorial` (below "`axioms`") only gives a contract: the procedure is guaranteed to terminate and in the final state the output parameter `result`[2] contains the value fac(n) where fac is the algebraically specified factorial-function. The second description of procedure `factorial` is found under "`declaration`". It defines a recursive implementation that describes how to calculate the result. To keep the specification consistent one would of course use only one of the two descriptions for a procedure.

KIV allows one to specify Abstract State Machines (ASM), a very general specification formalism for software [3,11]. These use a collection of abstract programs ("rules") to specify the steps of a transition system. To support them, programs may also contain parallel function assignments $f_1(t_1) := t_1', f_2(t_2) := t_2'$ and (infinite) nondeterministic choice:

choose x **with** φ **in** α **ifnone** β.

The construct binds x to any value satisfying φ , and executes α; if no value satisfies φ , β is executed.

The standard software development strategy in KIV is step-wise development of systems from abstract specifications to implementations. Different variants of Refinement (Data Refinement, Algebraic Refinement) are supported, the most common is ASM refinement, where ASMs model systems on different abstraction layers. This approach was successfully applied on a variety of verification projects, from compiler verification [27], over electronic payment systems [13,28] to the current verification of a file system [7,22,30].

$$\frac{\Gamma, A, B \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta} \qquad \frac{\Gamma \vdash A, \Delta \qquad \Gamma \vdash B, \Delta}{\Gamma \vdash A \wedge B, \Delta}$$

Fig. 4 Calculus rules for conjunction left and right

2.3 Concurrent systems in KIV

To specify concurrent systems, KIV extends the imperative language with interleaved programs. A variant of Interval Temporal Logic [20] called RGITL [2,29] is implemented to reason about both safety and liveness properties of such programs. The rules of the Rely/Guarantee calculus [16,37] are derivable in RGITL; therefore, Rely/Guarantee reasoning can be used as a proof principle. The logic was used for example in the analysis of lock-free algorithms [6,33,34].

2.4 Proving with KIV

KIV uses a Gentzen-style sequent calculus [32]. A sequent $\Gamma \vdash \Delta$ is a shorthand for $\forall \bar{x}. \bigwedge \Gamma \rightarrow \bigvee \Delta$ where $\bar{x}$ is the vector of all free variables in the lists Γ and Δ of formulae. Therefore, the variables in a sequent are always implicitly universally quantified. The rules of the calculus (and, therefore, the proof construction) follow the structure of the formulae. Figure 4 shows the primitive calculus rules for conjunction. The rules are applied backwards to reduce a proof obligation (the conclusion) to simpler premises.

Applying calculus rules step by step constructs a proof tree for the initial theorem. The proof tree is an explicit object in KIV, also represented graphically in KIV's GUI. Figure 5 shows a partial proof tree. Proof trees are stored when a proof attempt is finished; they can be loaded, manipulated (e.g. remove a subtree at a given position), investigated (e.g. which rewrite rules were used by the simplifier), and replayed. Replaying a proof or a part of a proof means re-executing all the (part's) proof steps in the same order. This is very helpful when a change to a theorem made its proof invalid as a whole but some parts are still valid and, therefore, can be reused in a new proof attempt. In this case, the still valid parts of the old proof are replayed at the appropriate goals in the new proof.

The information stored in proof trees is important for the correctness management, too (see Sect. 2.6).

The main proof technique for *wp*-calculus, i.e., for program correctness, in KIV is "symbolic execution". Sym-

[2] The semicolon in the parameter list separates input and reference parameters, so that assignments to the result in the body of the procedure are visible to the caller.

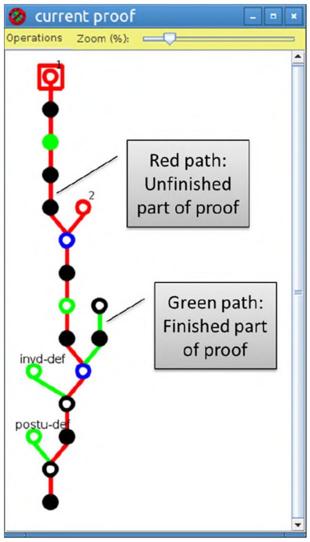

Fig. 5 Proof tree window

$$\frac{}{\{\mathrm{post}_x^t\}\ x := t\ \{\mathrm{post}\}} \qquad \frac{\{\mathrm{pre}_x^{x_0} \wedge x = t_x^{x_0}\}\ \alpha\ \{\mathrm{post}\}}{\{\mathrm{pre}\}\ x := t; \alpha\ \{\mathrm{post}\}}$$

Hoare's assignment rule   Rule for symbolic execution

Fig. 6 Assignment rules

bolic execution means that while proving, the program statements are sequentially executed and removed while calculating strongest postconditions from the preconditions. This is done until the program is removed and the remaining sequent is just a higher-order proof obligation. Symbolic execution replaces Hoare's assignment rule with a forward-reasoning version (the rule on the right in Fig. 6; a version of this rule appears already in [8]).

The rule is given for a nonempty α: the premise uses a new variable x_0 to store the value of x before the assignment. For an empty α, the premise is just the implication $\mathrm{pre}_x^{x_0} \wedge x = t_x^{x_0} \rightarrow \mathrm{post}$. Although the rule looks more complicated it is actually easier to use as it steps forwards through the program. We consider reasoning forwards more natural than Hoare-style reasoning as it basically corresponds to stepping through the program with a debugger.

For while-loops and recursion, invariants and induction are used. Besides the classical invariant-rule for while-loops (for total correctness some term must decrease according to a well-founded order), KIV's *wp*-calculus offers some specialized proof rules that shorten proofs:

- Rules for unwinding and exiting the while loop when the test evaluates to true resp. false
- An Ω-rule for while loops that replaces the loop with a finite number $n : nat$ of iterations of the loop body and allows a verification style with intermittent assertions as proposed in [5,19], that is sometimes more flexible than invariants:

$$\frac{\Gamma \vdash \exists n.\langle \mathbf{loop}\ (\mathbf{if}\ \varepsilon\ \mathbf{then}\ \alpha)\ \mathbf{times}\ n\rangle(\varphi \wedge \neg\ \varepsilon),\ \Delta}{\Gamma \vdash \langle \mathbf{while}\ \varepsilon\ \mathbf{do}\ \alpha\rangle\ \varphi,\ \Delta},$$

where $\mathbf{loop}(\alpha)\mathbf{times}\ n$ iterates the program α n times.

Although KIV is an interactive verification system, a high degree of automation is necessary to make industry-sized verification projects feasible. Therefore, the KIV system supports the proof engineer with a powerful simplifier. Simplification is based on applying the standard predicate logic rules with zero or one premise (case splits are implemented by separate heuristics) together with user-defined rewrite rules. These are specifically tagged theorems fulfilling certain syntactic conditions.

KIV's simplifier is quite sophisticated. Proofs are not constructed on the level of basic calculus rules ("conjunction left", "equivalence right", etc., though this option is available for educational purposes). Instead, the simplifier exhaustively applies all rewrite rules as well as propositional simplification, e.g. with the rule *conjunction left*, in a single step. This allows the proof engineer to focus on the really challenging tasks.

Simplification is context-aware, i.e., additional knowledge from the context of a subterm or subformula is taken into account. For example, when simplifying ψ (to some ψ') in the implication $\varphi \rightarrow \psi$, the simplifier assumes φ to be true. Conversely φ is simplified afterwards under the assumption $\neg\psi'$.[3] Building up a context for simplification of subformulas is similar to "grind" or the type-check in PVS [21] or to window inference [26] in Isabelle. Compared to PVS, KIV's simplifier does not use weakening rules, i.e., it always preserves provability of the goal. However, it does not integrate decision procedures or model checking. Compared to the Isabelle strategy, simplification is not applied only once to each subexpression, but until all formulas are maximally simplified with respect to their contexts (proofs in KIV never call the simplifier twice in a row).

The KIV simplifier applies all rewrite rules modulo associativity and commutativity (AC) of operators, which satisfy these properties. For example, because of the commutativity $n + m = m + n$, a rewrite rule $f(o) + l = \tau$ can also be applied to terms of the form $l + f(o)$. Another feature of the KIV simplifier is the support of conditional rewrite rules of the form $\varphi \rightarrow \tau = \tau'$. The substitution of (instances of) τ with τ' is only done when the (corresponding instance of the) precondition φ follows from the context of τ.

[3] Note that one needs to carefully track polarities of formulas here, as for example skolemization of $\exists$ is only possible for positive polarity.

KIV translates simplifier rules to functional pattern-matching code which gets compiled lazily upon usage to further improve speed [24]. Using these techniques the KIV simplifier is highly efficient and can work with tens of thousands of rewrite rules simultaneously. Compilation of simplifier rules into functional code is similar to the approach used in Isabelle [1], but KIV compiles all rewrite rules, in particular conditional (higher-order) ones as well as rewriting modulo associativity and commutativity.

The simplifier is combined with heuristics that encode established proof principles to substantially reduce the proof effort. Examples for heuristics are "structural induction" that attempts to prove a goal using structural induction for generated data types, and "symbolic execution" for automatically applying trivial rules for programs, such as the assignment rule. In order to work efficiently with KIV, knowledge about the simplifier is crucial. Without a proper use of the different types of simplifier rules and the experience when to use what kind of simplification, proofs for real problems require substantial user interaction and easily become intractable.

Rewrite rules are mostly used by the simplifier but they can also be applied interactively by the proof engineer. This is made easy by KIV's context-sensitive rewriting: the proof engineer simply clicks on a (sub-) term he wants to rewrite in KIV's graphical user interface (see Sect. 2.5) and the system offers a selection of all applicable rewrite rules.

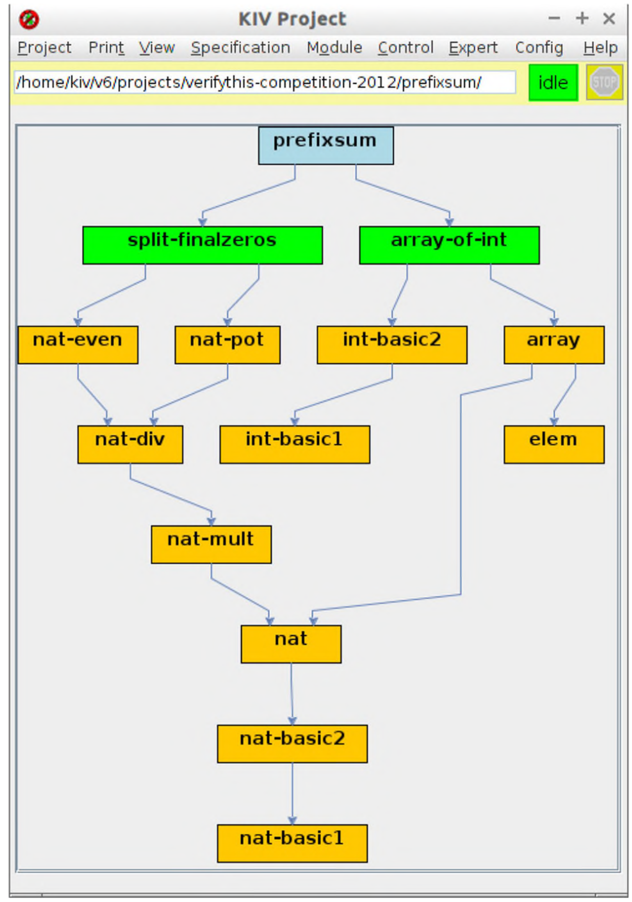

Fig. 7 Window with structured algebraic specification

2.5 KIV's graphical user interface

In contrast to most other interactive theorem provers, KIV offers an elaborate graphical user interface [12]. KIV's GUI has different windows for tasks like creating and changing a structured algebraic specification, editing specifications and theorems, managing the theorem base of a specification and, most important, the windows containing the proof tree and the current proof goal. All proof work in KIV is done by interacting with the GUI. KIV, unlike many other interactive provers, does not use proof scripts; there is no need to write, read or maintain textual representations of proofs.

Figure 7 shows a structured algebraic specification. Orange rectangles represent specifications imported from the library, green rectangles are specifications where all theorems are proven and the proofs are valid, and the blue rectangle is a specification with unfinished proof work.

Figure 5 shows an incomplete proof tree. Form and color of nodes indicate relevant information about the corresponding proof step. Filled circles indicate automatic proof steps, circles with holes are interactive steps. Black nodes are normal calculus rule applications, and blue circles represent used theorems and case distinctions and so forth. The proof tree contains all information about the current proof; it can be inspected by clicking on nodes and requesting specific information, such as the applied rule, used simplifier rules, and used substitutions. The proof tree can also be used to manipulate the proof by cutting of a subtree or attempting to replay the proof steps of a subtree. Having the proof tree as an interactive object is paramount to proving with KIV. Inspecting the proof steps that lead to the current goal is often important to understand the current proof obligation. This is especially true when the current goal is the result of a number of case distinctions and subsequent simplification.

Figure 8 shows the KIV window with the current proof goal. The large area to the right contains the sequent of the current proof goal. On the upper left side of the window is the list of applicable calculus rules. Proof steps are done by either selecting a step from the list on the left or by context-sensitive proving: Right-clicking somewhere into the current goal opens a small window with a list of all calculus rules and rewrite rules that are applicable on the sub-term the user clicked on. This can be seen on the right side of Fig. 8. Also helpful is the list of recently used theorems; they are collected on the lower left side and can be applied on the current goal by simply clicking on their name. On the top is a bar with buttons to easily switch between the different open goals, take back recent proof steps, and select and activate heuristics.

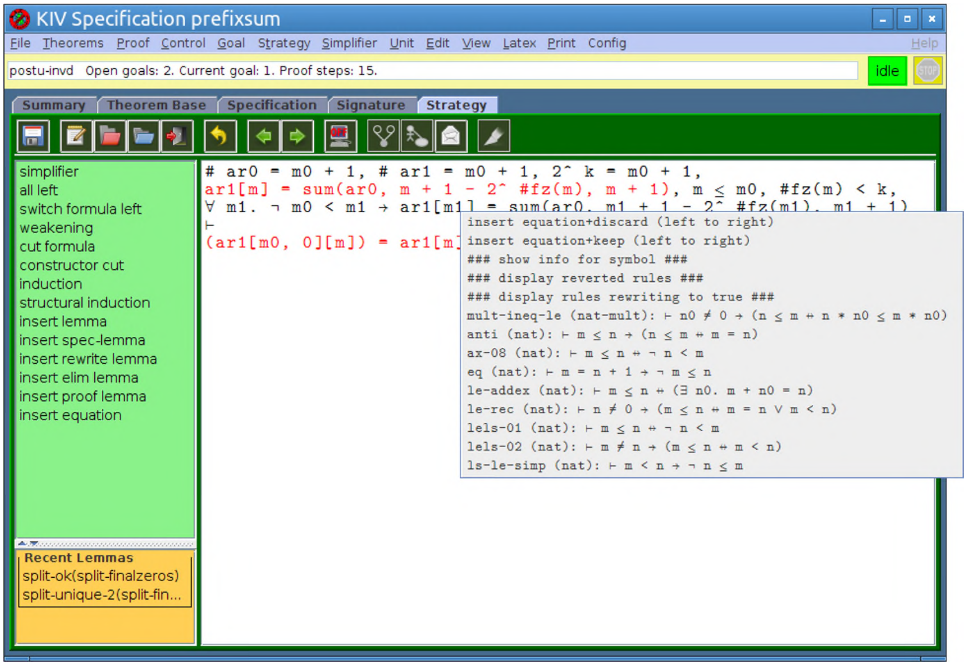

Fig. 8 Window with current goal, applicable rules, and context-sensitive Rewriting

2.6 Correctness management

When verifying a system, the proof engineer is most of the time disproving non-theorems. Definitions are too weak, preconditions missing, and so on. Therefore, theorems are bound to change a lot until the verification is finally finished. This is acceptable only if the amount of other theorems and proofs affected by a change is minimal. In order to keep the verification results correct, everything that depends on a modified theorem, axiom, etc. has to be analyzed to check if it is still valid. Keeping the proof correct while minimizing the results of changes is the responsibility KIV's elaborate correctness management [24].

KIV explicitly keeps track of signature symbols that appear in theorems and also of theorems that are used in proofs. A complete or partial proof is saved and contains all used lemmas and all signature symbols (functions, constants, procedures, etc.) that appear in the proof. When a specification is changed, by either adding or removing symbols, axioms, or theorems KIV checks which dependent specifications and proofs are no longer valid. Adding signature entries or axioms is never a problem, but removing them potentially makes dependent parts invalid. When a function symbol is deleted, all theorems and axioms that used the now-missing symbol are syntactically invalid. Modification or deletion of a theorem invalidates proofs that are directly dependent. A theorem with an invalid proof remains syntactically correct and can still be used in other proofs. Thus, invalidation of theorems does not propagate transitively, i.e., proofs that do not use the modified/deleted theorem remain valid. On the other hand, the correctness management does keep track of the transitive dependencies between theorems and their proofs to prevent cyclic proof dependencies.

Only changes to local theorems (i.e., theorems defined within the same node in the specification graph) result in an immediate invalidation of proofs that use this theorem. When a lemma in a dependent specification (e.g. from the library) is changed a theorem which uses the lemma does not become invalid directly, as this would prevent restructuring specifications or moving the lemma to another specification, without rendering theorems invalid. To guarantee that all proofs of a specification are valid, KIV uses an explicit "proved state" for specifications. Upon entering the proved state, KIV checks if all lemmas from dependent specifications used in the proofs are still present and marks them there as being used. Modifying the lemma in the library will then take the specification with the theorem out of the proved state.

2.7 Higher-order encoding of separation logic

Separation logic [25] facilitates the verification of pointer-based programs, such as the algorithm on binary trees given as the third challenge at the competition (see Sect. 3.3). A formula of Separation Logic is built from the following primitives: the assertion *emp* that the heap is empty, the maplet $r \mapsto o$ characterizing the singleton heap containing only the reference r : *ref* with object o : *obj* and the separating conjunction $P * Q$. The latter asserts that the heap consists of two disjoint parts where the formulae P and Q hold, respectively.

Traditionally, the heap is part of the semantics of separation formulas. In KIV, however, all parameters of a procedure—including the heap—are explicit. Heaps are specified as a non-free data type generated from the empty heap $\emptyset$, allocation of a reference r (written $h +\!\!+\, r$) and update of an object o at address r, written $h[r, o]$. $h[r]$ and $h \text{ -- } r$ denote dereferencing and deallocation, respectively. These are defined recursively over the structure of heaps. The type of the objects stored in the heap can be instantiated as needed for each case study.

As a consequence of having explicit heaps, predicates of type $heap \to bool$ (heap predicates) are used to encode Separation Logic formulae. In a sequent one can combine these with formulae of dynamic logic. Total correctness assertion (1), for example, states that starting with an initial heap satisfying the heap predicate P, the procedure α terminates and yields a heap that satisfies Q:

$$P(h) \vdash \langle\!|\alpha(;h)|\!\rangle\, Q(h) \tag{1}$$

The Separation Logic constants such as emp and connectives such as $*$ are introduced as higher-order functions mapping a fixed number of heap predicates to one. The separating conjunction for example has the following type, since it is a binary connective:

$$. * . : (heap \to bool) \times (heap \to bool) \to heap \to bool$$

As a higher-order function the maplet $r \mapsto o$ has the type $(ref \times obj) \to heap \to bool$ and is defined as follows:

$$(r \mapsto o)(h) \leftrightarrow h = (\emptyset +\!\!+\, r)[r, o] \wedge r \neq null$$

Furthermore, the KIV library supplies an overloaded version of quantifiers and all predicate logic connectives such as

$$. \wedge . : (heap \to bool) \times (heap \to bool) \to heap \to bool$$
$$\text{where} \quad (P \wedge Q)(h) \equiv P(h) \wedge Q(h)$$

so that the types work out.

It also provides instantiations of the heap for often-used data-structures such as lists and trees. For trees, for example, the generic sort obj stored in the heap is instantiated with the structure $node$. A node contains pointers to both the left and right subtrees and stores an additional value of a generic sort $elem$, i.e., it is generated by

$$node(left : ref,\ val : elem,\ right : ref)$$

Usually it is easier to reason about an abstract datatype than about the concrete pointer-based implementation. Therefore, the KIV library provides abstraction relations between common data-structures and their algebraic counterparts. We normally use a higher-order function that maps a value t of the algebraic data type to a heap predicate that

$$\frac{\Gamma_h^{h_0}, (r \mapsto o' * P)(h),\ h = h_0[r, o'],\ h_0[r] = o \vdash \langle\alpha\rangle\varphi, \Delta_h^{h_0}}{\Gamma, (r \mapsto o * P)(h) \vdash \langle h[r] := o';\ \alpha\rangle\ \varphi, \Delta}\ (:=_h)$$

Fig. 9 Rule for heap assignment

only holds if the heap contains a valid pointer-based representation of t.

For example, the KIV library has a free data type $tree$, generated by the constructors $leaf$ and $branch$:

$$tree = leaf \mid branch(left : tree,\ val : elem,\ right : tree)$$

The relation to the pointer-based binary tree is described by the function tr defined by structural recursion over the algebraic tree as follows (similar to [25]):

$$\begin{aligned} tr(r, leaf) &\equiv (emp \wedge \ulcorner r = null \urcorner) \\ tr(r, branch(t_0, a, t_1)) &\equiv \exists\, r_0, r_1.\ r \mapsto node(r_0, a, r_1) \\ &\qquad * tr(r_0, t_0) * tr(r_1, t_1) \end{aligned} \tag{2}$$

Leaves are represented by $null$ references. $\ulcorner\varphi\urcorner$ denotes the lifting of a pure formula $\varphi : bool$ (one that is independent of the heap, with $\ulcorner\varphi\urcorner(h) \equiv \varphi$, again to make the types work out). In the recursive case the existential quantifier binds two references, one for the root of each subtree.

In our formalism the heap is a single, explicit variable and modified by ordinary assignments. The heap update $h[r] := o$ is in fact just an abbreviation for $h := h[r, o]$. In principle, programs could also execute non-local assignments, such as $h := \emptyset$, or use several heaps, copying one over the other (using two heaps is unavoidable for refinement, but sometimes also beneficial in one program, to avoid explicit typing constraints for the elements). As a consequence, the general frame rule is invalid. As a substitute, we add an explicit higher-order variable $F : heap \to bool$ as a frame assumption to the contracts of programs. This means, for example, that instead of contract (1) above, we specify

$$(P * F)(h) \vdash \langle\!|\alpha(;h)|\!\rangle (Q * F)(h)$$

When the contract is needed in a proof about program β it is, therefore, possible to substitute the frame assumption F with the additional, concrete context of β that is unmodified by α. This basically means that we have to prove the frame rule for each program individually. However, for case studies (such as challenge 3 at the competition) this had no overhead, since it is usually trivial to establish that a write does not affect F.

To facilitate deduction with this Separation Logic framework, we extended KIV with an additional rule for heap assignments. Figure 9 shows this new rule of the calculus. It introduces a fresh variable h_0 for the old heap. In contrast to the normal assignment rule—shown on the right-hand side of Fig. 6—it yields a new Separation Logic formula about

```
lcp(ar : Array⟨elem⟩, n0 : nat, n1 : nat; m : nat) {
  m := 0;
  while   n0 + m < #ar ∧ n1 + m < #ar
        ∧ ar[n0 + m] = ar[n1 + m]
  do {
    m := m + 1
}}
```

Fig. 10 Longest common prefix

the new heap h and discards the formula about h_0. Only the information about h_0 that is needed for the invertibility of the rule is kept. Having such a rule in the calculus leads to more readable sequents and a significantly higher degree of automation in a proof. Except for the new assignment rule, no changes to KIV's core were necessary to support Separation Logic.

3 Solutions

This section documents the experience during the VerifyThis competition at the Formal Methods (FM) conference in Paris 2012 and presents further details on the KIV proofs for the 3 challenges. The solutions for challenge 1 (first part) and 3 were worked out at the competition. All programs, specifications, and proofs for the challenges are available at our web presentation [17].

3.1 Suffix arrays and longest repeated substring

This subsection details our solution to the first challenge of competition. It consisted of three parts: first, a verification of the calculation of the longest common prefix of two strings. Second, showing the correctness of the construction of a suffix array, and finally, proving that the given algorithm for the computation of the longest repeated substring of a string is correct.

3.1.1 Longest common prefix

The first part of the first challenge was the definition and verification of a procedure `lcp` that returns the length of the longest common prefix of two subarrays of integers. Figure 10 shows our implementation of `lcp`. We generalized the problem to arrays of an arbitrary type *elem*. The subarrays begin at index n_0 and n_1 in the array ar, respectively, and extend to the end of ar. The length of an array ar is written $\# ar$.

We proved that with a precondition of

$$n_0 < \# ar \ \wedge \ n_1 < \# ar$$

the postcondition $\varphi_1 \wedge \varphi_2$ holds, where φ_1 and φ_2 are defined as

$$\begin{aligned}\varphi_1 = {} & n_0 + m \leq \# ar \ \wedge \ n_1 + m \leq \# ar \\ & \wedge (\, \forall n.\, n < m \rightarrow ar[n_0 + n] = ar[n_1 + n]\,) \\ \varphi_2 = {} & (\, n_0 + m < \# ar \ \wedge \ n_1 + m < \# ar \\ & \rightarrow ar[n_0 + m] \neq ar[n_1 + m]\,)\end{aligned}$$

The first part φ_1 states that `lcp` returns an m within the bounds of each subarray and that the first m elements of the two subarrays are equal. The second part φ_2 asserts that this m is the largest one with this property, i.e., if the two subarrays extend beyond m, the values at index $m + 1$ are distinct.

The only interaction required from the proof engineer for this proof is the application of the invariant rule for the while-loop with the precondition and φ_1 as a loop-invariant and the termination measure $\# ar - m$. The rest of the proof is fully automated by heuristics that symbolically execute the program, by simplification of the parts of the formula that are concerned with arithmetic and by a heuristic that instantiates quantifiers.

We completed the implementation, specification, and proof of the first part of this challenge in approximately 10–15 min.

3.1.2 Construction of suffix arrays

The second part of the challenge consisted of the verification of a simple implementation of suffix arrays [18]. We needed too much time to understand and specify the algorithm during the competition and did not reach the actual verification; this solution has been developed after the competition.

Figure 11 shows our formalization of the algorithm `csa` that initializes the suffix array sa from the original array ar. The suffix array sa of an array ar stores the lexicographical order of all non-empty suffixes of ar as follows: let $suf(ar, i)$ be the suffix of ar starting at index i. Then, $sa[n] = i$ iff $suf(ar, i)$ is ranked at position n in the lexicographic order of all suffixes of ar.

In KIV, arrays are constructed from the uninitialized array $mkarray(n)$ of size n and updates of an array ar at position n with element e, written $ar[n, e]$. After allocation of the suffix array, its size is never changed.

The first while loop initializes the array sa so that afterwards the set of elements stored is $\{0, \ldots, \# ar - 1\}$. We define the function $elems_n(ar)$, which yields the multiset of all elements in the array ar up to, but excluding the element at index n. Thus, the invariant of the loop can be expressed as

$$elems_n(ar) = \{0, \ldots, n - 1\}. \tag{3}$$

With multisets, updating an element in the array propagates over $elems$ as removal of the old element and addition of the

```
csa(ar:Array⟨elem⟩; sa:Array⟨nat⟩) {
  sa ≔ mkarray(#ar);
  let n = 0 in {
   while n < #sa do {
     sa[n] ≔ n;
     n ≔ n + 1
  }};
  sort(λ i,i'.  suf(ar,i) <lex suf(ar,i')
             ∨ suf(ar,i) = suf(ar,i')
       ; sa)
}

sort(⊑ : (T × T → bool); ar : Array⟨T⟩) {
  let n = 0 in {
   while n < #ar do {
    let m = n in {
     while m > 0 ∧ ¬ ar[m - 1] ⊑ ar[m] do {
       ar ≔ ar[m, ar[m - 1]][m - 1, ar[m]];
       m ≔ m - 1
     };
     n ≔ n + 1
}}}}
```

Fig. 11 Construction of a suffix array

updated element. Sets do not satisfy this property, because in general one element may be stored at multiple indices.

After the initialization, the suffix array is sorted with a generic insertion sort, `sort`.[4] It is parametrized by the type T of elements and a parameter $\sqsubseteq$: $T \times T \to bool$ that defines an ordering relation.

T is instantiated to type *nat* of array indices, and $\sqsubseteq$ gets the reflexive closure of the lexicographic order on suffixes of ar starting at a given index. The predicate $<_{\text{lex}}$ is defined by

$$\begin{aligned}
&ar_1 <_{\text{lex}} ar_2 \equiv\\
&\quad \exists\, n.\, n \leq \#ar_1 \wedge n \leq \#ar_2\\
&\quad \wedge (\forall i < n.\, ar_1[i] = ar_2[i])\\
&\quad \wedge (\#ar_1 = n \to \#ar_2 > n)\\
&\quad \wedge (\#ar_1 \neq n \to \#ar_2 \neq n \wedge ar_1[i] < ar_2[i]),
\end{aligned}$$

i.e., ar_1 is lexicographically smaller than ar_2 if there is some valid index n of the first smaller element. Proving transitivity and totality of $<_{\text{lex}}$ (as required by `sort`) took approximately 50 manual steps.

Under the assumption that $\sqsubseteq$ is reflexive, transitive, and total, we establish the following postcondition of `sort`:

$$elems_{\#ar}(ar) = elems_{\#ar'}(ar') \wedge sorted(ar', \sqsubseteq) \tag{4}$$

where ar and ar' is the array before resp. after the call. The predicate $sorted(ar, \sqsubseteq)$ is defined by

$$\forall\, i, i'.\, i < i' \wedge i' < \#ar \to ar[i] \sqsubseteq ar[i']$$

The predicate $sorted(ar, i, i', \sqsubseteq)$ asserts that the subarray starting at index i (inclusive) and up to but excluding index i' is sorted and is defined similarly.

The multiset of elements of the array is invariant over both loops of the sorting algorithm and one requires only a trivial lemma showing that exchanging two elements of the array leaves the multiset unaltered.

For the outer loop the invariant unsurprisingly additionally asserts that the array is sorted up to the loop counter n, i.e., that $sorted(ar, 0, n, \sqsubseteq)$ holds. The postcondition of the procedure immediately follows from this invariant for $n = \#ar$.

The inner while loop lets the next element at index n trickle down as far as necessary to sort the array's first $n+1$ elements. The invariant of the inner loop, therefore, is

$$\begin{aligned}
&sorted(ar, 0, m, \sqsubseteq) \wedge sorted(ar, m, n+1, \sqsubseteq)\\
&\wedge (m \neq 0 \wedge m < n \to ar[m-1] \sqsubseteq ar[m+1]),
\end{aligned}$$

where m denotes the current position of the trickling element. The first line ensures that the array ranges $[0, m]$ and $[m, n+1]$ are sorted. According to the second part the neighboring elements of m are in order, implying that excluding m the suffix array is sorted up to n.

The verification of sorting required 15 manual steps, most of them to prove

- After exchanging two consecutive elements at index $m-1$ and m, the upper half $[m-1, n+1]$ of the array is sorted.
- After the inner while loop terminates, the two sorted array ranges can be merged into a single sorted range to establish the invariant of the outer while loop.

The postcondition of the procedure `csa` states that the suffix array is valid with respect to the original array, i.e., all suffixes of the original array are represented and the suffix array is sorted lexicographically.

$$\begin{aligned}
&post_{csa} \equiv\\
&\quad elems_{\#sa}(sa) = \{0, \ldots, \#ar - 1\}\\
&\qquad \wedge (\forall\, i, i'.\, i < i' \wedge i' < \#sa\\
&\qquad\qquad \to suf(ar, sa[i]) <_{\text{lex}} suf(ar, sa[i']))
\end{aligned}$$

The postcondition $post_{csa}$ follows from (3) and (4) with the help of the following two observations. First, two suffixes of an array are equal if and only if the starting indices are equal. Second, each index is stored only once in the suffix array. 35 interactions were necessary to derive this postcondition.

[4] This takes up a proposal of a reviewer.

```
lrs(ar : Array⟨elem⟩; i : nat, len : nat) {
  i := 0; len := 0;
  let n = 1, sa = [?] in {
    csa(ar; sa);
    while n < #ar do {
      lcp(ar, sa[n], sa[n - 1]; m);
      if m > len then {
        i := sa[n]; len := m
      };
      n := n + 1
    }
}}
```

Fig. 12 Longest repeated substring

3.1.3 Longest repeated substring

As the third part of the first challenge the organizers provided an algorithm that computes the longest repeated substring of an array by combining the other two procedures. Figure 12 shows our specification of the procedure `lrs`. It first creates a suffix array sa of the original array ar. Afterwards, the algorithm traverses the lexicographical order of the suffixes and in each iteration stores the index i and length len of the longest repeated substring found so far. If a prefix of a suffix is a longest repeated substring, then one of the neighbors in the lexicographical order can serve as a witness for the repetition. Therefore, the algorithm only calculates the longest common prefix of neighboring suffixes to determine whether i and len need an update.

For this part it is convenient to define an algebraic function $lcp(ar, n_0, n_1)$ returning the length of the longest common prefix of the subarrays starting at n_0 and n_1, respectively. We assume as a precondition that array's length is at least 1. The postcondition $post_{lrs}$ of `lrs` then states that there is a witness $i' \neq i$ for the repetition of the string starting at index i of length len. Furthermore, all other repetitions (j, j') are not longer:

$$\begin{aligned} post_{lrs} \equiv\ & i < \#ar \\ & \wedge (\exists i'.\, i' \leq \#ar \wedge i \neq i' \wedge lcp(ar, i, i') = len) \\ & \wedge (\forall j, j'.\ j \leq \#ar \wedge j' \leq \#ar \wedge j \neq j' \\ & \qquad \rightarrow lcp(ar, j, j') \leq len) \end{aligned}$$

Note that we allow the empty suffix at the end of the array as a valid witness. The invariant of the while loop adds that the suffix array in the local variable sa is valid, i.e., $post_{csa}$ holds and restricts the repetitions to those found in the n lexicographically smallest suffixes and to those where the witness of the repetition is the immediate successor:

$$\begin{aligned} & post_{csa} \wedge \#ar \neq 0 \wedge n \neq 0 \wedge n \leq ar \wedge i < \#ar \\ & \quad \wedge (\exists i'.\, i' \leq \#ar \wedge i \neq i' \wedge lcp(ar, i, i') = len) \\ & \quad \wedge (\forall j.\ j+1 < n \rightarrow lcp(ar, sa[j], sa[j+1]) \leq len) \end{aligned}$$

On entering the loop, the witness $i' = \#ar$ is used to establish the existence of a repetition of length 0.

It is also easy to prove that the invariant is maintained by an iteration of the loop: only one pair of suffixes $(sa[n], sa[n+1])$ is added and needs to be considered.

The difficult case is establishing the postcondition after the loop terminates. The following lemma is used by the proof:

$$\begin{aligned} & i_0 \leq \#ar \wedge i_1 \leq \#ar \wedge i_2 \leq \#ar \\ & \wedge suf(ar, i_0) < suf(ar, i_1) \wedge suf(ar, i_1) < suf(ar, i_2) \\ & \quad \rightarrow lcp(ar, i_0, i_2) \leq lcp(ar, i_0, i_1) \end{aligned}$$

It asserts that the lcp of neighboring suffixes is longer than that of suffixes further apart in the lexicographical order. It is proven by induction over $lcp(ar, i_0, i_2)$ and required about 25 interactions excluding some trivial simplifier rules and an additional lemma stating that if $ar[n_0] = ar[n_1]$ and $suf(ar, n_0) < suf(ar, n_1)$ holds, so does $suf(ar, n_0 + 1) < suf(ar, n_1 + 1)$ requiring an additional 20 interactions. Another 10 interactions were needed to conclude that if we have a valid suffix array sa and a repeated substring at $sa[i_0]$ with witness $sa[i_1]$ and length len (with $i_0 < i_1$, the other case is similar), then $sa[i_0]$ with witness $sa[i_0 + 1]$ is a repeated substring with length at least len.

In total, we needed approximately two days to specify and verify the second and third part of the first challenge.

3.2 Prefix sum

The `prefixsum` algorithm modifies an array ar of integers that is assumed to have a power of two as its size. In the computed result each element $ar[i]$ contains the sum of all elements that were at indices 0 to $i - 1$ in the original array. The computation is done in two phases, using two auxiliary programs `upsweep` and `downsweep`. For both of them the task description gave a recursive and an iterative implementation as a Java program. The main difficulty for verifying the algorithm is to figure out what intermediate sums it computes. We could not find a solution during the challenge, so we worked out a solution offline. Since we immediately saw that the intermediate sums of the iterative version were based on a binary representation of the indices (see below), we verified the iterative version. To do this, we first translated this version to KIV syntax, as shown in Fig. 13.

Compared to the Java version of the task description, `upsweep` does not require an extra local variable “right”.

```
uph(left : nat, space : nat; ar : array[nat]) {
  while left < # ar do {
   ar[left + space] := ar[left] + ar[left + space],
   left := left + 2 * space
}}

upsweep(; ar : array[nat]) {
  let space = 1 in {
    while space < # ar do {
      uph(space - 1, space; ar);
      space := 2 * space
}}}

downh(right : nat, space : nat; ar : array[nat]) {
  while right < # ar do {
    let left = right - space,
        i = ar[right]
    in {
      ar[right] := ar[left] + i;
      ar[left] := i
    };
    right := right + 2 * space
}}

downsweep(space : nat; ar : array[nat]) {
  while space > 0 do {
    downh(2 * space - 1, space; ar);
    space := space / 2
}}

prefixsum(; ar : array[nat]) {
  upsweep(; ar);
  ar[# ar - 1] := 0;
  downsweep(# ar / 2; ar);
}
```

Fig. 13 Prefix sum in KIV syntax

Instead, it uses parallel assignments (separated by comma), just to demonstrate their use. Return statements have been dropped (the return value of `upsweep` is ignored anyway) or replaced with input/output parameters (recall, that the semicolon in calls and declarations separates input parameters from input/output parameters). As KIV's procedures forbid global variables, the global array is also an input/output parameter. The assignment at the start of `downsweep` has been moved to the top-level algorithm, as it logically does not belong to the `downsweep` algorithm. Auxiliary procedures were defined for the inner loops, just to be able to verify these as separate lemmas that do not have to repeat the text of the while loops.

Figure 14 shows how the algorithm works on an example. The first line labeled *init* shows some initial array contents. Lines uk for $k = 1, 2, 3$ show the array after k iterations of `upsweep`. Note that variable *space* at the start of each

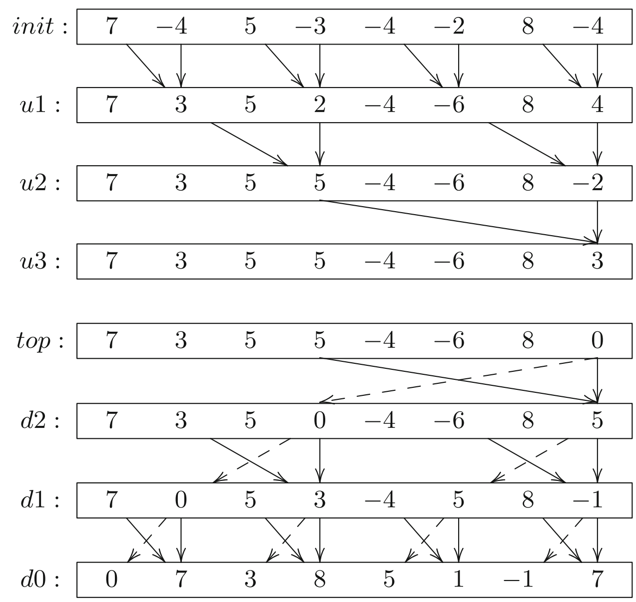

Fig. 14 How prefix sum works

iteration has the value 2^k. The arrows indicate how content is modified: each pair of solid arrows indicates that the entry below has been replaced with the sum of the entries above. If one numbers the indices from 1 to 8 (KIV uses zero-based indices from 0 to 7), then it is immediately obvious that the first iteration modifies even indices only, the second modifies indices which are a multiple of 4, and the last one requires the index to be a multiple of 8 (here: the last index only). Stated a bit more formally, a zero-based index n is modified by iteration k, iff $n + 1$ is a multiple of 2^k. This is equivalent to the binary representation of $n + 1$ having at least k final zero bits.

After `downsweep`, the `prefixsum` algorithm sets $a[7]$ to zero, before starting `downsweep` (line *top*). Lines dk for $k = 2, 1, 0$ show the array content after one, two, and all three iterations of this algorithm. The numbers are chosen such that again iteration k of `downsweep` has $space = 2^\wedge\ k$ at the start of each loop, and that entry $a[m]$ is modified, iff $m + 1$ has at least m final zero bits. Solid arrows are as before, and dashed arrows indicate that a value has been moved.

For verification, we express total correctness as

$$pre(ar, ar_0) \rightarrow \langle \texttt{prefixsum}(; ar)\rangle\, post(ar, ar_0)$$

in KIV, where

$$pre(ar, ar_0) \equiv ar = ar_0 \wedge (\exists\, k_0.\ \#ar = 2^\wedge\ k_0)$$
$$post(ar, ar_0) \equiv (\forall\, m < \#ar.ar[m] = sum(ar_0, 0, m))$$

are the pre- and postcondition of the algorithm. Function $sum(ar, m, n)$ is defined recursively to compute the sum of array elements between m (inclusive) and n (exclusive) for $m \leq n \leq \#ar$.

To figure out the relevant invariants, we have taken a top-down approach. First, we declared predicates $invu$, $invuh$, $invd$, $invdh$, and $postu$ that express the invariants of the outer/inner loops of `upsweep` and `downsweep`, and the postcondition of `upsweep`. Without giving any axioms for these predicates, we derived the following contracts for each of the procedures. The first two connect upsweep and downsweep by $postu$, (5) and (6) establish $invu$ resp. $invd$ as loop invariants for `upsweep` and `downsweep`.

$$pre(ar, ar_0) \rightarrow \langle \mathtt{upsweep}\,(;ar)\rangle\, postu(ar, ar_0)$$

$$postu(ar_1, ar_0) \wedge ar = ar_1[\#ar - 1, 0] \rightarrow \langle \mathtt{downsweep}\,(space; ar)\rangle\, post(ar, ar_0)$$

$$invu(space, ar, ar_0) \wedge space < \#ar \rightarrow \langle \mathtt{uph}\,(left, space; ar)\rangle invu(2 * space, ar, ar_0) \tag{5}$$

$$invd(space, ar, ar_0) \wedge space \neq 0 \rightarrow \langle \mathtt{downh}\,(right, space; ar)\rangle invd(space\,/2, ar, ar_0) \tag{6}$$

The program proofs then can be reduced to a number of predicate logic goals using symbolic execution. KIV computes the following sufficient conditions for `upsweep`:

$$pre(ar, ar_0) \rightarrow invu(1, ar, ar_0) \tag{7}$$

$$invu(space, ar, ar_0) \wedge space < \#ar \rightarrow invuh(space - 1, space, ar, ar_0) \tag{8}$$

$$invuh(left, space, ar, ar_0) \wedge left < \#ar \rightarrow invuh(left + 2 * space, space, ar[left + space, ar[left] + ar[left + space]], ar_0) \tag{9}$$

$$\neg\, left < \#ar \wedge invuh(left, space, ar, ar_0) \rightarrow invu(2 * space, ar, ar_0) \tag{10}$$

$$\neg\, space < \#ar \wedge invu(space, ar, ar_0) \rightarrow postu(ar, ar_0) \tag{11}$$

Similarly, the conditions for `downsweep` are

$$postu(ar_1, ar_0) \wedge ar = ar_1[\#ar - 1, 0] \rightarrow invd(\#ar/2, ar, ar_0) \tag{12}$$

$$invd(space, ar, ar_0) \wedge space \neq 0 \rightarrow invdh(2 * space - 1, space, ar, ar_0) \tag{13}$$

$$invdh(right, space, ar, ar_0) \wedge right < \#ar \rightarrow invdh(right + 2 * space, space, ar[right, ar[right - space] + ar[right]][right - space, ar[right]], ar) \tag{14}$$

$$\neg\, right < \#ar \wedge invdh(right, space, ar, ar_0) \rightarrow invd(space\,/2, ar, ar_0) \tag{15}$$

$$invd(0, ar, ar_0) \rightarrow post(ar, ar_0) \tag{16}$$

Formulas 7 and 12 establish the invariant of the loop in `upsweep` resp. `downsweep` from the precondition. Formulas 8 and 13 establish the invariants of the helper procedures' loops. That the invariants are maintained over the loops is proven by Lemma 9 and 14 for the inner loops and 10 and 15 for the outer loops. Afterwards, the postcondition is established by formula 11 and 16, respectively.

These lemmas are immediately applied as rewrite rules to close all goals of the program proofs (except some trivial termination goals).

The strategy then was to iteratively figure out the invariants by doing proof attempts for the goals above using tentative definitions. This led to several corrections until the goals were finally provable. Inspecting proof trees of failed proof attempts to find out which corrections were necessary was (as in all case studies) of invaluable help in this iterative process. The invariance goals (9) and (14) for the inner loops are the hardest to prove. They were done last, as attempts with invariants that fail to prove the other (simpler) goals above would just waste time.

The hard part for our proof then was to figure out which sums `upsweep` and `downsweep` exactly compute. As indicated above, it was clear that the range was somehow based on a binary representation of the index values. Therefore, we defined a function $\#fz$, such that $k = \#fz(m)$ is the number of final zero bits of the binary representation of $m + 1$.

$$\#fz(2 * m) = 0 \quad \#fz(2 * m + 1) = \#fz(m) + 1$$

As an afterthought, we recognized that $\#fz(m)$ also computes the number of final 1-bits of the original index m.

3.2.1 Upsweep

The sum $S(m, k)$ that `upsweep` computes in $ar[m]$ for a zero-based index m then is

$$S(m, k) \equiv sum(ar_0, m + 1 - 2^\wedge\ k, m + 1),$$

where k increases through the loop starting with $k = 0$ ending with $\#fz(m)$ (space is always equal to $2^\wedge\ k$). Note that if m is even (i.e., if we have an odd one-based index n), the sum is just $ar_0[m]$.

The definition uses powers of two (written as prefix $2^\wedge$) from KIV's library (these were originally defined for a case study on dynamic hash tables [9] that also uses arrays whose size is a power of two) axiomatized as

$$2^\wedge\ 0 = 1 \qquad 2^\wedge\ (n + 1) = 2 * 2^\wedge\ n$$

The definition of $invu$ for the main loop of `upsweep` thus is defined as

$$\begin{aligned}
&invu(space, ar, ar_0) \equiv \\
&\exists\, k, k_0.\ \#ar = \#ar_0 \wedge \#ar = 2^\wedge\ k_0 \\
&\quad \wedge\ k \le k_0 \wedge space = 2^\wedge\ k \\
&\quad \wedge\ (\forall\, m < \#ar.\ ar[m] = S(m, min(k, \#fz(m))))
\end{aligned}$$

The postcondition for `upsweep` is simply the special case where the minimum is just $\#fz(m)$.

$$\begin{aligned}
&postu(ar, ar_0) \equiv \\
&\quad \#ar = \#ar_0 \wedge (\exists\, k_0.\ \#ar = 2^\wedge\ k_0) \\
&\quad \wedge\ \forall\, m < \#ar.\ ar[m] = S(m, \#fz(m))
\end{aligned}$$

The invariant $invuh$ for the inner loop is derived from the outer invariant, by discerning the case whether the array index m has already been considered ($left < m$) or not (we write the case split, using KIV's if-then-else operator. $\supset$.; .). In the latter case $ar[m]$ still contains the sum for k; otherwise, already the sum for $k + 1$

$$\begin{aligned}
&invuh(left, space, ar, ar) \equiv \\
&\exists\, k, k_0.\ \#ar = \#ar_0 \wedge \#ar = 2^\wedge\ k_0 \\
&\quad \wedge\ k < k_0 \wedge space = 2^\wedge\ k \\
&\quad \wedge\ (\exists\, n.\quad odd(n) \wedge left = n * space - 1 \qquad (*) \\
&\qquad\qquad \wedge\ left < \#ar + 2 * space) \qquad\qquad (*) \\
&\quad \wedge\ (\forall\, m < \#ar.\ ar[m] = \\
&\qquad\quad S(m, min(m < left \supset k + 1; k), \#fz(m))
\end{aligned}$$

The difficult part for this invariant was to figure out the lines marked with ($*$) that define the exact constraints needed for $left$.

3.2.2 Downsweep

The invariant for `downsweep` is also difficult to determine. The value of $space$ is always of the form $(2^\wedge\ k)\ /2$ for some k.[5] Indices m of the array, where $\#fz(m) < k$ have not been modified by `downsweep`, their value still is equal to the result $S(m, \#fz(m))$ of `upsweep`. The difficult case are indices with $\#fz(m) \ge k$. After some experiments with concrete values, we found that these store sums from zero up to $m + 1 - 2^\wedge\ k$. For the final $k = 0$ this is the sum from zero to m that should be computed. Thus the invariant $invd$ is

$$\begin{aligned}
&invd(space, ar, ar_0) \equiv \\
&\exists\, k, k_0.\ \#ar = \#ar_0 \wedge \#ar = 2^\wedge\ k_0 \\
&\quad \wedge\ k \le k_0 \wedge space = 2^\wedge\ k\ /2 \\
&\quad \wedge\ (\forall\, m < \#ar.\ ar[m] = \\
&\qquad\quad (\#fz(m) < k \supset S(m, \#fz(m)); \\
&\qquad\qquad\qquad\qquad sum(ar_0, 0, m + 1 - 2^\wedge\ k)))
\end{aligned}$$

Given the invariant for `downsweep` the invariant $invdh$ for the inner loop again needs a case split, whether the index has been modified or not. Indices m with $\#fz(m) < k$ are not touched by the inner loop, and indices with $\#fz(m) = k$ have not been modified, when $m + 2 * space > right$. Both still contain the value computed by `upsweep`. Indices m with $\#fz(m) \ge k$ that have been passed by the loop ($m + 2 * space \le right$) have been modified to the new sum with an incremented k; all others still store the sum from the previous iteration. Altogether this gives

$$\begin{aligned}
&invdh(right, space, ar, ar_0) \equiv \\
&\exists\, k, k_0.\quad \#ar = \#ar_0 \wedge \#ar = 2^\wedge\ k_0 \\
&\quad \wedge\ k < k_0 \wedge space = 2^\wedge\ k \\
&\quad \wedge\ (\exists\, n.\quad n \ne 0 \wedge right = 2 * n * space - 1 \\
&\qquad\qquad \wedge\ right < \#ar + 2 * space) \\
&\quad \wedge\ \forall\, m < \#ar.\ ar[m] = \\
&\qquad (\quad (\quad \#fz(m) = k \wedge m + 2 * space > right \\
&\qquad\qquad \vee \#fz(m) < k) \\
&\qquad\quad \supset S(m, \#fz(m)); \\
&\qquad\quad (\quad m + 2 * space \le right \\
&\qquad\qquad \supset sum(ar_0, 0, m + 1 - 2^\wedge\ k); \\
&\qquad\qquad\ sum(ar_0, 0, m + 1 - 2^\wedge\ (k + 1)))))
\end{aligned}$$

Figuring out initial guesses for the invariants $invu$, $invuh$, and $invd$ required several hours of thinking. Setting up specifications, and reducing proofs to predicate logic goals required about one hour of work, iterative proof attempts for the goals resulting from the verification of `upsweep` took a good day of work until they were successful. The final proofs all start with four trivial interactions: two to expand the definitions, one to instantiate the existential quantifier for k and k_0 of the postcondition with the values from the preconditions and one to get the appropriate array index m for the universal quantifier. All final proofs are simple (15 interactions or less), except the two main invariance proofs (9) and (14) which need 39 and 52 interactions.[6]

---

[5] Note that the k used in the formula is one higher than the number used in Fig. 14. This means that k is 0 instead of -1 at the end of the loop. Thus, we can stay within natural numbers.

[6] Since these proofs were done last, they were not done as separate lemmas, but as part of the proof of (5) and (6).

Some information about required lemmas is (15) relies on the fact, that $m + 2 * space > right$ implies that the last of the three cases of $invdh$ is not possible, which requires a bit of reasoning about powers of two. (12) and (8) both rely on the fact that the only number m with $m < 2^\wedge n$ that satisfies $\#fz(m) \geq n$ is $m = 2^\wedge\ n - 1$ (and $\#fz(m) = n$ for this m). Finally, the two difficult invariance proofs (9) and (14) crucially depend on the fact that any number $n + 1$ can be uniquely split into $\#fz(n)$ many final zero bits and an odd number $lead(n)$, representing the leading bits. Defining

$$lead(2 * n) = 2 * n + 1 \qquad lead(2 * n + 1) = lead(n)$$

the following two lemmas are easy to prove by well-founded induction:

$$n + 1 = lead(n) * (2^\wedge\ \#fz(n))$$
$$odd(n_0) \to (\quad (2^\wedge\ n) * n_0 = (2^\wedge\ \#fz(m)) * lead(m)$$
$$\leftrightarrow n = \#fz(m) \wedge n_0 = lead(m))$$

Both proofs (9) and (14) need a number of trivial facts about odd and even numbers, e.g. that $odd((2^\wedge\ n) * lead(m))$ is true only for $n = 0$ (since $lead(m)$ is always odd). Indeed the last correction for the definition of $invuh$ (which was rather difficult to find) was to add the conjunct $odd(n)$.

A number of interactions are finally spent on appropriately rewriting goals by applying distributivity. From our experience, distributivity is the main source of difficulty for proofs with multiplication, since it must be applied in both directions (automatic rewriting in one direction is a bad idea) to make suitable other rules (e.g. cancelation of summands in equations or inequations; or substitution of one equation into another) applicable. Rewriting modulo associativity and commutativity (as built into KIV's simplifier) can only help to infer suitable matches, when applying the rule manually, but does not help with automation.

Altogether the case study proves 35 theorems and lemmas with 279 interactions for 541 proof steps (excluding approx. 600 lemmas about natural numbers and arrays imported from KIV's library) with an overall effort of 2 days for one person.

3.3 Delete the minimum of a binary search tree

The third challenge consisted of the verification of the procedure `delete_min`, which removes and returns the minimum of a non-empty binary search tree. Our specification is shown in Fig. 15: The algorithm uses three references l, p, and pp to traverse to the leftmost node of the tree.

We use the embedding of separation logic into higher-order logic described in Sect. 2.7 instantiated for heaps of $nodes$. We assume a strict total order $<$ on $elem$.

Furthermore, we employed the tr predicate to specify that the nodes in the heap, starting with the root r, correspond to an algebraic tree t. This abstraction motivated splitting the verification effort into two parts: first a correspondence between the actual program and operations on the algebraic tree is proven. Afterwards, the desired correctness criterion is derived solely by reasoning about the algebraic tree.

Part 1. To capture the effect of the program on the algebraic tree t, we defined the functions $t.leftmost$ and $t.butleftmost$ via structural recursion over t. The former returns the leftmost element of t, while the latter yields t with its leftmost element removed. The effect of the *butleftmost* function on the algebraic tree corresponds to the effect of the actual algorithm on the pointer-based tree, i.e., the same rotation is performed to remove the leftmost element. We prove

$$(tr(r, t) * F)(h), r \neq null$$
$$\vdash \langle \texttt{delete\_min}(; m, r, h)\rangle\ post,$$

where

$$post \equiv \quad m = t.leftmost \wedge (tr(r, t.butleftmost) * p)(h)$$

Given that r points to the root of a non-empty tree represented by t via heap-predicate tr (see Sect. 2.7) the program and the algebraic operations compute the same result (and the arbitrary frame F stays intact).

The proof proceeds by symbolic execution of the call (replacing it by the implementation of `delete_min`). The case $p = null$ is trivial. The interesting aspect of the proof is the loop in the resulting (schematic) goal

$$r = pp, (tr(r, t) * F)(h) \vdash \langle loop; modification\rangle\, post$$

Intuitively, any loop invariant must specify that "executing the modification after the loop leads to the postcondition". Now observe that termination of the loop can be derived from the *abstract* tree t, i.e., in each iteration pp is represented by

```
delete_min(; m:elem, r:Ref, H:Heap[node]) {
  let p = H[r].left in {
    if p = null then {
      m := H[r].elem;
      r := H[r].right, H := H--r
    } else let pp = r, l := H[p].left in {
      /* loop */
      while l ≠ null do {
        pp := p; p := l; l := H[p].left
      };
      /* modification */
      m := H[p].elem; l := H[p].right;
      H := H -- p;
      H[pp].left := l
    }}}
```

Fig. 15 Delete the minimum of a binary search tree

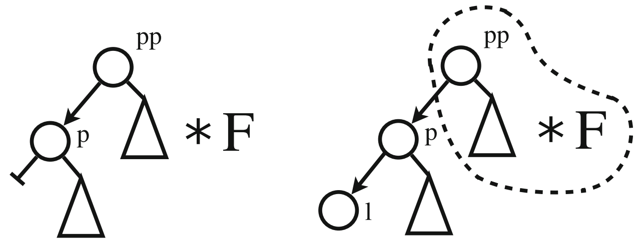

Fig. 16 Base and inductive case of the deletion

some smaller algebraic tree. This argument suggest well-founded induction over the size $\#t$ of t (counting constructor terms = nodes) instead of a normal invariant. We get the following induction hypothesis as an additional assumption (where t_0 designates the initial tree):

$$\forall F, h, r, t.\ \big(\# t < \# t_0 \land (tr(r, t) * F)(h) \\ \rightarrow \langle loop; \mathit{modification}\rangle\, post\big)$$

In the base, where $l = null$, the loop exits immediately. Figure 16 shows the heap before the subsequent pointer modifications removing the node p. Since the algebraic rotation is exactly the same, the goal becomes trivial.

The recursive case needs to execute the loop once, thereby unfolding the tree abstraction. This yields the heap assertion for h

$$tr(p, t.\mathit{left}) * \underbrace{pp \mapsto node(p, t.val, r) * tr(r, t.right) * F}_{\text{new frame}}$$

The remaining iterations of the loop and the modifications afterwards are covered by the induction hypothesis.

The proof is automated nicely: the correct instance of the induction hypothesis is found automatically by the quantifier instantiation heuristics of KIV with the help of the simplifier's pattern matching modulo associativity and commutativity for multiple $*$-conjuncts of the shape of the heap. Concretely, the frame F is mapped to the previous frame plus the root pp of the current tree and its right subtree, designated by the dotted part of the right diagram in Fig. 16; the tree t becomes $t.\mathit{left}$ and the root r becomes $p = h[pp].\mathit{left}$.

In the inductive case the validity of the postcondition can be inferred automatically. In the base case unfolding the tr predicate (manually) is sufficient. In total, the proof has five simple interactive proof steps.

During the competition the predicate tr was unfolded too eagerly, leading to sequents that were more complex than necessary and hampering automation. Afterwards, we disabled the corresponding simplifier rule and achieved the above-mentioned degree of automation.

Part 2. The above, however, does not prove that the minimal element is returned and removed from the tree. Only for this property it is necessary to assume that the tree is a search tree. We defined $a \in t$ for an element a and an algebraic tree t recursively and then specified the search tree property $bst(t)$ for a branch of the tree as follows:

$$bst(branch(t_0, a, t_1)) \\ \leftrightarrow\ (\forall b.\, b \in t_0 \rightarrow b < a) \land (\forall b.\, b \in t_1 \rightarrow a < b) \\ \land bst(t_0) \land bst(t_1)$$

For leaves bst always holds. Assuming $t \neq leaf$ and $bst(t)$ we then showed the following:

1. $bst(t.\mathit{butleftmost})$
2. $t.\mathit{leftmost} \in t$
3. $\forall b.\, b \in t.\mathit{butleftmost} \rightarrow t.\mathit{leftmost} < b$

Thus, the resulting tree is again a binary search tree, the returned element is minimal and removed from the tree. The latter property is only valid, because we assumed that elements are strictly ordered. A final theorem combined all these results into one postcondition for the program.

During the competition we assumed garbage collection and added an additional $(\cdots * true)$ to the postcondition to capture that the program produces nodes that are unaccounted for by the tree afterwards and the explicit frame assumption p. While cleaning up the solutions we realized that all the proofs can be automatically replayed for a version with explicit memory management.

The solution was completed almost in time during the competition. The reason for the delay was actually not the first part involving separation logic, but the second part, which required a significantly higher amount of manual interactions than our final version presented here. In preparation for this paper, we spent approximately half an hour cleaning up and simplifying the solution.

Extensions. A reviewer suggested a more abstract specification of the problem in terms of ordered sets. The web presentation now demonstrates a data refinement from sets with strictly ordered elements, exploiting the abstraction to algebraic trees as intermediate layer.

Ordered sets s as specified in the KIV library are equipped with operations $a \in s$, $s.min$ and $s.butmin$. Given a mapping from trees to sets $t.set = \{a \mid a \in t\}$ the correspondence of these operations to $t.\mathit{leftmost}$ and $t.\mathit{butleftmost}$ is easily established with the help of search tree properties 2 and 3 and a similar characterization of the minimal element in a set. Refinement of sets to the pointer structure then follows. We have additionally implemented and proven (non-rebalancing) insertion, which took us about half an hour.

4 Comparison

This sections compares the KIV solutions the ones of other teams based on the material available on the competition website[7] and, of course, based on the authors' personal opinion.

Challenge 1. The LCP problem was the easiest in the competition and rather straightforward to solve. For example, the teams using Why3, KeY, and GNATprove show contracts that are almost identical to ours.

There is a Java solution and a C solution for LRS in VeriFast, both abstracting to algebraic lists. The Java solution has helper definitions to quantify over the elements of the list and some associated lemmas, relying on VeriFast's support for recursive predicates. The C version specifies LCP over the algebraic abstraction and proves essentially a simulation property. The program is annotated with several explicit lemma invocations (according to the documentation, VeriFast does not do much proof search). Part of the extra effort can be attributed to the complexity of real programming languages, in particular, to memory management.

The KeY and Why3 teams developed solutions to LRS after the competition as well.

Challenge 2. Other teams verified the recursive version of the prefix sum algorithm. We learned after the competition that the recursive version is more interesting than the iterative one, since it is the basis for parallelization of the algorithm. They abstracted the computation on the array to a binary algebraic tree, where the leaves represent the initial array contents and each inner node a recursive call that sums the contents of its children and writes them in the rightmost array field. This abstraction could be chosen such that each of the two recursive calls establishes the abstraction for the left resp. right child in the tree. The remainder of the call then establishes the abstraction for the parent node. This solution is very elegant and was found by the VeriFast team during the competition using Separation Logic. Imitating this solution in KIV would require a separation library for arrays instead of heaps.

We do not think that this solution can be adapted to the iterative version of the algorithm, because the computation iterates over the array several times and in each round a different increment is used. Thus, each round basically establishes the next-higher layer of the tree abstraction. Only at the end the entire tree abstraction holds.

Challenge 3. The VeriFast solution to the tree challenge is probably the closest to ours. Based on separation logic they define an abstraction to algebraic trees that records the exact pointers as well, instead of existentially quantifying them (compare our definition (2)). This should lead to less quantifier instantiations in principle (although these have not been a problem for us). They characterize the search tree property by a predicate

$$ordered\_between(min, t, max)$$

recursively over the algebraic tree t, which is slightly easier to reason about than the combination of two recursive definitions $\in$ and bst in our solution.

As far as we know, no other team used or could use a direct induction on the algebraic tree for wp-formulas instead of a loop invariant. This is actually a technique we use regularly in case studies, in particular when refinement requires to consider several programs simultaneously.

Other teams encoded the modifications explicitly in the loop invariant. Techniques to do so include the *magic wand* operator $-\!\!*$ from separation logic that cuts out sub-parts of the heap. Apparently none of the tools in the competition supported this operator directly, although the VeriFast team could emulate it. A solution with $-\!\!*$ would be interesting to investigate in the future (the operator is already formalized in the KIV library).

Block contracts [35] are another approach to specify this particular invariant elegantly; it is, in fact, quite similar to our solution.

The KeY team used dynamic frames which required a couple of helper definitions. Their solution was developed after the competition and relies on model methods (a new feature of KeY), which are essentially logical functions and predicates. In contrast to our approach, the KeY team abstracted to a flat sequence (representing the inorder traversal) instead of a tree and proved that, abstractly, its first element is removed.

Termination resp. well-foundedness of model methods in the KeY solution is established by an invariant on a ghost field `height` in each tree node tracking the maximum height of its subtrees, whereas in KIV and VeriFast this fact is derived from the finiteness of the algebraic tree.

Another solution using "zippers" was developed after the competition by the Why3 team.

General remarks. In summary we think that the set of features provided by a tool strongly influences the approach to a solution. For example, we note that we (as well as the KeY team) use quantifiers liberally due to the fact that we can provide instances interactively if the heuristics fails to find the correct one. In contrast, our impression is that the VeriFast approach avoids quantifiers in favor of denoting witnesses constructively, in conjunction with a form of existential variables in assertions that seems to be susceptible to automation. A related question is whether to use auxiliary (ghost) state or logical functions to capture abstract values.

We observe that in any case abstraction capabilities play a central role and we believe that the need for proof automation

[7] http://verifythis2012.cost-ic0701.org/solutions.

should not overly limit expressiveness of the specification language.

It would be desirable to come up with a more in-depth comparison how *development* of solutions is supported (i.e., automated tests, counterexamples, debugging of failed proofs) in contrast to an analysis of the final results. One approach would be to collect data such as screen recordings during the competition itself, as suggested in [15].

Finally, it would be interesting to compare how different approaches scale up to larger problems, but that is probably outside the scope of such competitions. Based on our experience we estimate that such larger challenges will put more focus on features like dependency management and proof debugging and reuse.

5 Conclusion

In this paper we presented the interactive theorem prover KIV and our solutions to the problems posed at the VerifyThis competition 2012. Compared to the previous competition [4], the difficulty of the problems had a bigger variation. Especially the two algorithms on suffix arrays and the prefix sum computation were hard to understand in the limited time allotted for their verification. A complication for our team was that some of the procedures were rather long and given as Java source code. Hence, we required quite some time to specify them in KIV.

Clear strengths of KIV are the expressiveness of its logic and the library of algebraic data types and their associated theorems. This facilitates modular proofs, as, for example, for the third challenge: we were able to first abstract the problem to an algebraic data type and then infer the correctness in a second proof from the abstract properties. Furthermore, with KIV it was possible to establish this abstraction elegantly with an induction over the size of the tree. This was only possible by leveraging the features of dynamic logic, which is not limited to an invariant proof for loops. The versatility of higher-order logic also made it possible to embed separation logic into KIV, without explicit prior support. The logic also facilitates verifying abstract programs, e.g. the generic elements for the tree in challenge 3 or the order $\sqsubseteq$ only given by its properties in challenge 1. Another distinguishing feature of KIV is the interactive user interface, facilitating the debugging of failed proof attempts.

However, we had some difficulties with arithmetic and with arrays. There we could benefit from integration of decision procedures, such as SMT solvers. The time spent on failing proof attempts could be reduced by the counterexamples given by an SMT solver or an integration of Alloy [36].

We have not used the Java calculus implemented in KIV, as it basically adds overhead to the proofs: since Java's type system (with inheritance) is not encodable into the type system of the logic, type constraints need to be handled as logic formulas when using the Java calculus. This would have required extra deduction effort instead of just using type checking. Also, all data structures are stored in the heap in Java, while for the abstract programs no such embedding is required. Finally, another source of overhead is exception handling: array access would have to check out-of-bounds exceptions, while the algebraic function $a[i]$ will just yield some unspecified integer (which of course will also make goals unprovable, if an out-of-bounds access exists in the code). Nevertheless, adding partial functions (that are just used in programs) or a predicate type system as in PVS could make the gap between abstract programs and concrete code smaller. Altogether, we currently prefer a refinement-based approach for bigger case studies (see, e.g. [10] for the Mondex case study), that automatically translates abstract code to Java and proves a 1:1 refinement using a simple mapping from abstract data to low-level representation.

The participation at the competition led to a small enhancement in our earlier rule for heap assignments to increase the degree of automation of proofs involving Separation Logic. Otherwise, the system was used as is. As a result of the competition we have recently added assertions and contracts to specifications (as opposed to just defining theorems). This will make it easier for others to understand the general idea of the proof without having to inspect the actual theorems and their proof tree. Additionally, this information will be used by heuristics, reducing manual user interaction. After seeing the elegant VeriFast solution for the prefix sum algorithm, we also plan to integrate Separation Logic for arrays (to split subranges of arrays by the $*$-operator). In addition to figuring out suitable rewrite rules this would probably require some minor adjustments to the assignment rule.

Acknowledgments We thank the organizers for providing interesting problems and for the opportunity to discuss the solutions with the other participants after the competition. For us, this led to a better understanding of the difficulties involved and how they were tackled by individual teams and their tools. We also thank the reviewers for their constructive reviews. They have led to small improvements and extensions for two of the case studies.

References

1. Aehlig, K., Haftmann, F., Nipkow, T.: A compiled implementation of normalization by evaluation. In: Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics (THOL 08), vol. 5170, pp. 39–54. Springer LNCS, Berlin (2008)
2. Bäumler, S., Schellhorn, G., Tofan, B., Reif, W.: Proving linearizability with temporal logic. FAC J **23**(1), 91–112 (2011)
3. Börger, E., Stärk, R.F.: Abstract state machines—a method for high-level system design and analysis. Springer, Berlin (2003)
4. Bormer, T., Brockschmidt, M., Distefano, D., Ernst, G., Filliatre, J. C., Grigore, R., Huisman, M., Klebanov, V., Marche, C., Monahan, R., Mostowski, W.I., Poiikarpova, N., Scheben, C., Schellhorn, G.,

Tofan, B., Tschannen, J., Ulbrich, M.: The cost ic0701 verification competition 2011. In: International Conference on Formal Verification of Object-Oriented Software, FoVeOOS 2011, vol. 7421, pp. 3–21. Springer, Berlin (2012)
5. Burstall, R.M.: Program proving as hand simulation with a little induction. Inf. Process. **74**, 309–312 (1974)
6. Derrick, J., Schellhorn, G., Wehrheim, H.: Mechanising a correctness proof for a lock-free concurrent stack. In: Proceedings of FMOODS 2008, Oslo, vol. 5051, pp. 78–95. Springer LNCS, Berlin (2008)
7. Ernst, G., Schellhorn, G., Haneberg, D., Pfähler, J., Reif, W.: Verification of a virtual filesystem switch. In: Proceedings of Verified Software: Theories, Tools, Experiments, vol. 8164, pp. 242–261. Springer, Berlin (2014)
8. Floyd, R.W.: Assigning meanings to programs. In: Schwartz, J.T. (ed.) Proceedings of the Symposium on Applied Mathematics, vol. 19, pp. 19–32. American Mathematical Society, New York (1967)
9. Fuchß, T., Reif, W., Schellhorn, G., Stenzel, K.: Three Selected Case Studies in Verification. In: Broy, M., Jähnichen, S.: (eds.) KORSO: Methods, Languages, and Tools for the Construction of Correct Software—Final Report, vol. 1009. Springer LNCS, Berlin (1995)
10. Grandy, H., Bischof, M., Schellhorn, G., Reif, W., Stenzel, K.: Verification of Mondex Electronic Purses with KIV: From a Security Protocol to Verified Code. In: FM 2008: 15th International Symposium on Formal Methods, vol. 5014. Springer LNCS, Berlin (2008)
11. Gurevich, Y.: Evolving algebras 1993: Lipari guide. In: Börger, E. (ed.) Specification and validation methods, pp. 9–36. Oxford Univ. Press, Oxford (1995)
12. Haneberg, D., Bäumler, S., Balser, M., Grandy, H., Ortmeier, F., Reif, W., Schellhorn, G., Schmitt, J., Stenzel, K.: The user interface of the KIV verification system—as system description. Electronic Notes in Theoretical Computer Science UITP special issue (2006)
13. Haneberg, D., Moebius, N., Reif, W., Schellhorn, G., Stenzel, K.: Mondex: engineering a provable secure electronic purse. Int. J. Softw. Inf. **5**(1):159–184 (2011). http://www.ijsi.org
14. Harel, D., Kozen, D., Tiuryn, J.: Dynamic logic. In: Gabbay, D., Guenther, F., (eds.) Handbook of philosophical logic, vol. 4, , 2nd edn, pp. 99–217. Kluwer Academic Publishers, Dordecht (2002)
15. Huisman, M., Klebanov, V., Monahan, R.: On the organisation of program verification competitions. In: Proceedings of the 1st International Workshop on Comparative Empirical Evaluation of Reasoning Systems (COMPARE), CEUR Workshop Proceedings, vol. 873 CEUR-WS.org (2012)
16. Jones, C.B.: Specification and design of (parallel) programs. In: Proceedings of IFIP'83, pp. 321–332. North-Holland, Amsterdam (1983)
17. Web presentation of the KIV solutions for the VerifyThis Competition, 2012. http://www.informatik.uni-augsburg.de/swt/projects/verifythis-competition-2012
18. Manber, U., Myers, G.: Suffix Arrays: a new method for on-Line string searches. In Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms, SODA '90, pp. 319–327. Society for Industrial and Applied Mathematics (1990)
19. Manna, Z., Waldinger, R.: Is "sometime" sometimes better than "always"?: intermittent assertions in proving program correctness. Commun. ACM **21**(2), 159–172 (1978)
20. Moszkowski, B.: A temporal logic for multilevel reasoning about hardware. IEEE Comput. **18**(2), 10–19 (1985)
21. Owre, S., Rajan, S., Rushby, J.M., Shankar, N., Srivas, M.: PVS: Combining specification, proof checking, and model checking. In: Alur, R., Henzinger, T.A. (eds.) Computer Aided Verification (CAV), vol. 1102. Springer LNCS, Berlin (1996)
22. Pfähler, J., Ernst, G., Schellhorn, G., Haneberg, D., Reif, W.: Formal specification of an erase block management layer for flash memory. In: Bertacco, V., Legay, A. (eds.) Hardware and software: verification and testing, vol. 8244, pp. 214–229. Springer International Publishing LNCS, Berlin (2013)
23. Reif, W., Schellhorn, G., Stenzel, K., Balser, M.: Structured specifications and interactive proofs with KIV. In: Bibel, W., Schmitt, P. (eds.) Automated Deduction—A Basis for Applications, vol. II, pp. 13–39. Kluwer, Dordrecht (1998)
24. Reif, W., Schellhorn, G., Stenzel, K., Balser, M.: Structured specifications and interactive proofs with KIV. In: Bibel, W., Schmitt, P. (eds.) Automated Deduction—A Basis for Applications, vol. II Systems and Implementation Techniques. Interactive Theorem Proving, Chap. 1, pp. 13–39. Kluwer Academic Publishers, Dordrecht (1998)
25. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: Proceedings of LICS, pp. 55–74. IEEE Computer Society, New York (2002)
26. Robinson, P., Staples, J.: Formalizing a hierarchical structure of practical mathematical reasoning. J. Logic Comput. **3**(1), 47–61 (1993)
27. Schellhorn, G., Ahrendt, W.: The WAM case study: verifying compiler correctness for prolog with KIV. In: Bibel, W., Schmitt, P. (eds.) Automated Deduction—A Basis for Applications, vol. III. Applications, Automated Theorem Proving in Software Engineering, Chap. 3, pp. 165–194. Kluwer Academic Publishers, Dordecht (1998)
28. Schellhorn, G., Grandy, H., Haneberg, D., Moebius, N., Reif, W.: A systematic verification approach for Mondex electronic purses using ASMs. In: Glässer, U., Abrial, J.-R. (eds.) Dagstuhl Seminar on Rigorous Methods for Software Construction and Analysis, vol. 5115, pp. 93–110. Springer LNCS, Berlin (2009)
29. Schellhorn, G., Tofan, B., Ernst, G., Reif, W.: Interleaved programs and rely-guarantee reasoning with ITL. In: Proceedings of the 18th International Symposium on Temporal Representation and Reasoning (TIME), pp. 99–106. IEEE Computer Society Press, New York (2011)
30. Schierl, A., Schellhorn, G., Haneberg, D., Reif, W.: Abstract specification of the UBIFS file system for flash memory. In: Proceedings of FM 2009: Formal Methods, vol. 5850, pp. 190–206. Springer LNCS, Berlin (2009)
31. Stenzel, K.: Verification of Java Card Programs. PhD thesis, Universität Augsburg, Fakultät für Angewandte Informatik, http://www.opus-bayern.de/uni-augsburg/volltexte/2005/122/ (2005)
32. Szabo, M.E. (ed.): The Collected Papers of Gerhard Gentzen. North-Holland, Amsterdam (1969)
33. Tofan, B., Schellhorn, G., Reif, W.: Formal verification of a lock-free stack with hazard pointers. In Proc. ICTAC, vol. 6916, pp. 239–255. Springer LNCS, Berlin (2011)
34. Tofan, B., Schellhorn, G., Reif, W.: Local rely-guarantee conditions for linearizability and lock-freedom. Karlsruhe Reports in Informatics 2011–26, Karlsruhe Institute of Technology (2011)
35. Tuerk, T.: Local reasoning about while-loops. In: Müller, P., Naumann, D., Yang, H. (eds.) Proceedings, VS-Theory Workshop of VSTTE 2010, pp. 29–39 (2010)
36. The Alloy Project. http://alloy.mit.edu
37. Xu, Q., de Roever, W.-P., He, J.: The rely-guarantee method for verifying shared variable concurrent programs. FAC J **9**(2), 149–174 (1997)