VSTTE 2010
Workshop Proceedings

Edited by

Rajeev Joshi, Tiziana Margaria, Peter Müller,
David Naumann, and Hongseok Yang

Preface

The third instance of *Verified Software: Theories, Tools and Experiments* hosted two workshops:

VS-Theory (18th August) focuses on theoretical foundations of software verification. Topics range from the difficult and essential study of soundness of delicate proof methods, to the discovery of new specification techniques and proof methods, to dramatic simplification or unification of existing methods, to as yet unknown breakthroughs. The workshop was co-chaired by David Naumann, Stevens Institute of Technology, and Hongseok Yang, Queen Mary, University of London.

VS-Tools & Experiments (19th August) focuses on the development of verification tools and their experimental evaluation. Topics of interest include interfaces between tools, tool integration platforms, and case studies. The workshop was co-chaired by Tiziana Margaria, University of Potsdam, and Rajeev Joshi, NASA/JPL Laboratory for Reliable Software.

The program committees thoroughly reviewed all submissions and selected five papers for presentation at the Theory workshop and six papers for the Tools & Experiments workshop. These proceedings contain the accepted papers of both workshops.

We would like to thank the authors of all submitted papers, whose work allowed us to compile an exciting workshop program. We are grateful to the program committee members for their careful reviewing and to the VSTTE organizers for their support.

Zurich, August 2010

Rajeev Joshi
Tiziana Margaria
Peter Müller
David Naumann
Hongseok Yang

Program Committees

VS-Theory

Amal Ahmed, Indiana University, US
Rajeev Alur, University of Pennsylvania, US
Anupam Datta, Carnegie Mellon University, US
Yannis Kassios, ETH Zurich, CH
Neel Krishnaswami, Microsoft Research, UK
Daniel Kroening, Oxford University, UK
Antoine Mine, CNRS, FR
Aleks Nanevski, IMDEA Software, SP
David Naumann, Stevens Institute of Technology, US (co-chair)
Tamara Rezk, INRIA, FR
Dave Schmidt, Kansas State University, US
Ashish Tiwari, SRI International, US
Viktor Vafeiadis, University of Cambridge, UK
Hongseok Yang, Queen Mary University of London, UK (co-chair)

VS-Tools & Experiments

Leo Freitas, U. York
Rajeev Joshi, NASA/JPL Laboratory for Reliable Software (co-chair)
Moonzoo Kim, Korea Advanced Institute of Science and Technology
Joe Kiniry,IT Uni of Copenhagen
Daniel Kroening, Oxford
Tiziana Margaria, University of Potsdam (co-chair)
Thomas Santen, Microsoft
Gerhard Schellhorn, Uni Augsburg
Bernhard Steffen, Uni Dortmund

Table of Contents

Local Reasoning about Mashups

Philippa Gardner, Gareth Smith, and Adam Wright

Imperial College London, UK
{pg, gds, adw07}@doc.ic.ac.uk

Abstract. Web mashups are complex programs that dynamically compose XML data and JavaScript code from many sources. Whereas data is sometimes formally specified by XML schema, code never is. This makes it difficult to construct reliable software. Using local Hoare reasoning, introduced in separation logic to reason about e.g. C programs and extended in context logic to reason about e.g. the DOM library, we are able to reason about mashup programs, proving that they are fault-free and providing specifications for code that are analogous to XML schema for data.

1 Introduction

The growing 'mashup' phenomenon involves websites using DOM and JavaScript to create complex web applications that integrate data and code from many sources. This leads to problems with reliability: a website may change unaware that remote code depends on it; clients may misuse a service due to lack of specification; or two sites may conflict due to incompatible use of resource. Building on the work specifying the W3C DOM library for XML update [1, 2], we show how to give specifications to mashup components. These specifications describe both the shape of the component's data (analogous to XML schema) and the behaviour of the code it provides (a kind of schema for code). With such specifications, one can construct provably fault-free mashup programs, where services deliberately expose a subset of their resource and clients ensure the integration of components is sound.

Our program reasoning is based on local Hoare reasoning, introduced in separation logic [3] to reason modularly about C programs and extended in context logic [4] to reason about program modules such as the DOM library. The work on DOM arose in two stages, by first giving a formal specification of Featherweight DOM [1] and recently giving a complete specification of the fundamental interfaces of DOM Core Level One [2]. This paper provides a first step to reasoning about programs which call the DOM library, by applying the featherweight approach to reason about mashup programs. In particular, we reason about Featherweight DOM combined with key features of JavaScript used in mashup programs: *mashup IO commands* for accessing remote data and code, and *dynamic functions* for sharing reusable, dynamic parametric code. Our dynamic functions precisely capture the dynamic nature of JavaScript functions, an essential ingredient of mashup programs. Full JavaScript functions are more complex, being truly higher-order and tightly integrated with JavaScript's object system. We do not work with higher-order functions or a full object system as our mashup application does not rely on these features and they would increase the complexity of this presentation. In particular, higher-order functions in local reasoning have so far proved to be particularly technically challenging [5]. Just as the work on Featherweight DOM extended to the fundamental interfaces of DOM Core Level 1, we hope the work presented here will provide a stepping stone to reasoning about substantial JavaScript programs which call the DOM library.

We illustrate our ideas using a mashup example based on the popular Twitter Service. Twitter is a service where users can share textual updates. Twitter exposes the update list as XML. Subsets of this Twitter data are often embedded within other websites: for example, our research group homepage might list all the recent Twitter updates people have targeted at us. Unfortunately, the update list can be polluted with spam, which could therefore appear on our homepage. One solution is to use a spam-checker service to filter known spammers from the list before we display it. This combination of three elements (our homepage, the Twitter data, and the spam filtering system) forms a mashup.

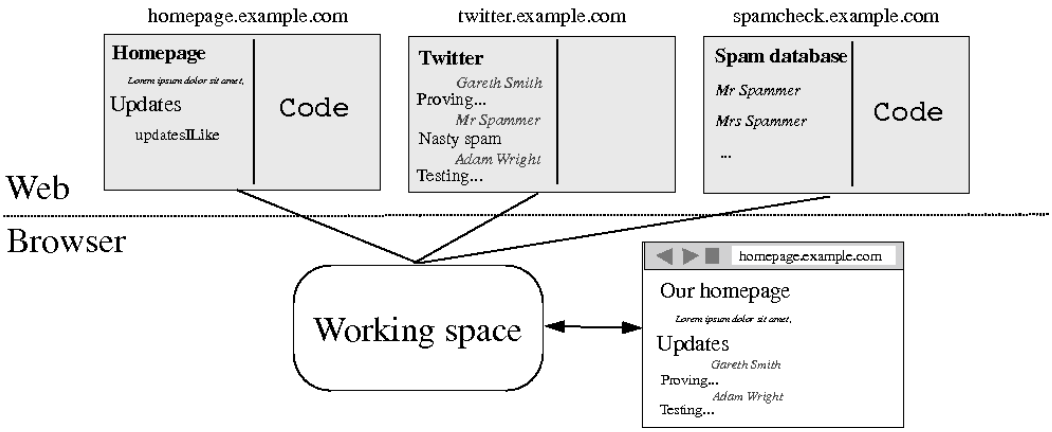


Fig. 1. Twitter mashup example

Figure 1 illustrates this Twitter mashup example. It shows a browser displaying homepage.example.com. The browser is directed by the code of homepage to get the data from homepage and Twitter, and data and code from the spam-checker, and form the mashup. In reality, the data and code will be web pages - a mixture of XML and JavaScript code, possibly spread over several files. In this paper, we work with data and code that has already been parsed by the browser into DOM data and separate code.

We reason about the homepage code. We assume published specifications for the data and code being pulled from the web. The specification of the Twitter data is that it is a tree whose children are a list of Twitter updates. The specification of the spam-checker is that its code requires a string (a person's name) as input and returns a boolean. With these assumptions and a suitable specification of the homepage data, we can prove that the homepage code is fault-free and that the only change to the homepage data is the addition of the Twitter updates. We require no additional assumptions about how the spam-checker works. In future, we envisage using a combination of trusted published specifications (say from a site such as Google), together with untrusted specifications which must be automatically checked, and unspecified data and code which require automatically-generated specifications.

Related work: There are several pieces of work which apply techniques from formal methods to DOM and JavaScript. The ideas in this paper build on Gardner, Smith, Wheelhouse and Zarfaty's previous work on formally specifying DOM [1, 2], which followed Thiemann's work on a type safe DOM [6]. Our language is very much inspired by features from JavaScript. Maffeis, Mitchell and Taly have given an operational semantics of JavaScript [7], which significantly aided our understanding of JavaScript. There is various work on adding XML schema-like types to programming languages in general (e.g. XDuce [8]), and JavaScript in particular (e.g. [9, 10]). We follow the view of Vitek [11], who has provided a systematic analysis of JavaScript programs on the web and has come to the conclusion that a type system for real world JavaScript is impractical. His solution is to work on a new programming language Thorn [12]. Our solution is to work with local Hoare reasoning.

There is a recent body of work which restricts the mashup environment to make it safe to put guest code on any web page: examples include Adsafe [13], Caja [14] and Subspace [15, 16]. As far as we are aware there is no published work on using logical specifications to prevent runtime faults, or determine mashup dependancies. There has been a wide body of work looking at reasoning about traditional program modules using separation logic (e.g. [17, 18, 19]). This reasoning is course-grained, in the sense that a client can reason about separate abstract data structures of the module (e.g. sets or DOM trees). In contrast, our context-logic reasoning is fine-grained, in the sense that a client can reason about subdata (e.g. subsets or DOM subtrees). See [20].

2 Reasoning about Featherweight DOM

The DOM API is a W3C standardised library [21] for manipulating tree data and is ubiquitous in web programming: for example, all browsers provide an implementation of DOM for manipulating web pages. Featherweight DOM, introduced by Gardner, Smith *et al.* [1], combines a simple imperative WHILE language with a fragment of the DOM node interfaces which concentrates on the XML tree structure. We summarise their work on using local Hoare reasoning based on context logic to provide an axiomatic semantics of Featherweight DOM.

Data structure: Featherweight DOM programs manipulate a standard variable store and a unique tree-shaped object heap, which reflects the structure of the documents being edited. The data structure defining our tree-shaped heap consists of documents, trees, forests (lists of children of a tree), groves (heaps of rooted trees) and strings:

$$\begin{aligned}
\text{Documents} \quad & \mathbf{doc} ::= \#\text{document}_{\text{id}}[\mathbf{de}]_{\text{fid}} \\
\text{Document Elements} \quad & \mathbf{de} ::= \langle \mathbf{t} \rangle_{\text{DE}} \mid \varnothing_{\text{DE}} \\
\text{Trees} \quad & \mathbf{t} ::= \mathbf{s}_{\text{id}}[\mathbf{f}]_{\text{fid}} \mid \#\text{text}_{\text{id}}[\mathbf{s}] \\
\text{Forests} \quad & \mathbf{f} ::= \varnothing_{\text{F}} \mid \langle \mathbf{t} \rangle_{\text{F}} \mid \mathbf{f} \otimes \mathbf{f} \\
\text{Groves} \quad & \mathbf{g} ::= \varnothing_{\text{G}} \mid \langle \mathbf{doc} \rangle_{\text{G}} \mid \langle \mathbf{t} \rangle_{\text{G}} \mid \mathbf{g} \oplus \mathbf{g} \\
\text{Strings} \quad & \mathbf{s} ::= \varnothing_{\text{S}} \mid \mathbf{c} \mid \mathbf{s} \cdot \mathbf{s}
\end{aligned}$$

Unlike DOM, we do not associate trees with a specific owner document, so the #document node just acts as a root container for a DOM tree. We do this only to give a compact presentation; it is simple to add owned trees to our data structure and adapt our reasoning accordingly [2]. Notice that, as well as a tree having an associated node identifier (id), its child list has a forest identifier (fid). These identifiers are analogous to heap addresses and must be unique across the entire data structure. This allows commands to address nodes directly, regardless of the current tree structure. Programs may traverse the structure using commands such as `getChildNodes`, which returns the forest identifier of a given node, or `item` which takes a forest identifier and an integer "n", and returns the "n"th node of the given forest. We also work with single-holed contexts for all these structures, defined as standard with the holes annotated with the types of the removed data: the document type DOC, document element type DE, tree type T, forest type F, grove type G and string type S.

We define an equivalence on this tree-structured data: the $\otimes$ operator is associative with identity $\varnothing_{\text{F}}$; $\oplus$ is associative and commutative with identity $\varnothing_{\text{G}}$; and $\cdot$ is associative with identity $\varnothing_{\text{S}}$.

With this structure, we can represent a simple version of the Twitter update list. Written in XML, it has the form

```
<twitter>
  <update><user>Adam Wright</user> Hi</update>
  <update><user>Gareth Smith</user> Hello</update>
</twitter>
```

In this paper, we do not work with XML directly but rather its"parsed" form, suitable for manipulation by our Featherweight DOM programs. For example, the above XML Twitter example has the parsed form:

$$\begin{aligned}
\text{twitterdata} \triangleq \#\text{document}_1[\text{twitter}_2[& \\
& \text{update}_3[\text{user}_4[\#\text{text}_5[\text{Adam Wright}]]_6 \otimes \#\text{text}_7[\text{Hi}]]_8 \otimes \\
& \text{update}_9[\text{user}_{10}[\#\text{text}_{11}[\text{Gareth Smith}]]_{12} \otimes \#\text{text}_{13}[\text{Hello}]]_{14} \\
&]_{15}]_{16}
\end{aligned} \tag{1}$$

using a shorthand for string notation ($\mathbf{F} \cdot \mathbf{o} \cdot \mathbf{o}$ is rendered as Foo) and omitting type-annotations such as $\langle\rangle_{\text{F}}$ when they are clear from the context. We also sometimes omit some unique identifiers, empty strings within text nodes, and empty child lists of tree nodes.

```
Statement ::=
    Id   = IdExpr | Str  = StrExpr
  | Int  = IntExpr | Bool = BoolExpr
  | if ( BoolExpr ) then { Statement }
                    else { Statement }
  | while ( BoolExpr ) { Statement }
  | skip | Command
  | Statement; Statement
```

```
Command ::=
    Id  = createElement(IdExpr, StrExpr)
  | Str = getNodeName(IdExpr)
  | Id  = getParentNode(IdExpr)
  | Id  = getChildNodes(IdExpr)
  | Id  = appendChild(IdExpr, IdExpr)
  | Id  = removeChild(IdExpr, IdExpr)
  | Id  = item(IdExpr, IntExpr)
  | ...
```

Fig. 2. Featherweight DOM Language

Language: The language of Featherweight DOM is given in figure 2.

With Featherweight DOM, we can write programs that construct and analyse DOM trees: for example,

```
updates = getChildNodes(node);
thirdKid = item(updates, 2);
appendChild(publishHere, thirdKid)
```

This program first obtains the list of children for the tree whose top node is identified by `node`. It then extracts the third child in this list, storing the identifier in `thirdKid`. It then moves the node identified by `thirdKid` to be the last child of the node identified by `publishHere`. This program will only run correctly on certain DOM trees. Such trees must contain a node identified by `node` which has at least three children. Moreover, the third child must not be an ancestor of the node identified by `publishHere`.

Assertion language: We define an assertion language to specify properties of the variable store and our tree-structured heaps. Recall that assertion languages based on separation logic consist of three parts: classical first-order logic assertions, separating assertions constructed from the commutative separating conjunction $*$ and its right adjoint $-\!\!*$, and heap-specific assertions such as $\varnothing$ denoting the empty heap and $x \mapsto y$ denoting a heap with exactly one cell. Here, we will use an assertion language based on context logic, which has a similar structure. In our case, the separating assertions are constructed from a non-commutative separating application $\circ_{\mathrm{D}}$, and its two separating right adjoints $\circ\!\!-_{\mathrm{D}}$ and $-\!\!\circ$. The assertion $P \circ_{\mathrm{D}} Q$ specifies that a given tree can be split into two disjoint parts, a subtree of type D satisfying Q and a tree context satisfying P. The assertion $P \circ\!\!-_{\mathrm{D}} Q$ specifies a tree such that, whenever a context satisfying P is placed around the given tree, then the result (of type D) satisfies Q. The assertion $P -\!\!\circ Q$ specifies a context such that, whenever a subtree satisfying P is put inside the given context, the resultsatisfies Q. We do not require a type annotation in the assertion $P -\!\!\circ Q$, since the type of the context satisfying $P -\!\!\circ Q$ implies the types of the data satisfying P and Q.

As our heap is given by a tree-shaped data structure, we give assertions that describe these trees. Our tree assertions are constructed by lifting the structure of the data. For example, we can use these assertions to define an *update* predicate that accepts only update nodes with a user node and #text node as children:

$$\mathit{update} \triangleq \mathrm{update}[\mathrm{user}[\#\mathrm{text}] \otimes \#\mathrm{text}] \tag{2}$$

As before, we omit angle brackets when the meaning is clear. We also omit identifiers and strings inside text nodes when they are existentially quantified, and the child lists of nodes when they are empty.

We have the following logical equivalence between assertions, demonstrating the use of the separating application:

$$\mathit{update} \iff \mathrm{update}[-_{\mathrm{T}} \otimes \#\mathrm{text}] \circ_{\mathrm{T}} \mathrm{user}[\#\mathrm{text}]$$

Using separating application, we can also derive the assertions $\Box_{\otimes} P$ and $\diamond_{\mathrm{D}_1 \to \mathrm{D}_2} P$. The assertion $\Box_{\otimes} P$ specifies a forest in which every tree at the top level satisfies P. The assertion $\diamond_{\mathrm{D}_1 \to \mathrm{D}_2} P$ specifies a tree or forest of type D_2 with a subtree somewhere, potentially deep down, of type D_1 that satisfies P. For example, we can compactly describe an arbitrary update list in Twitter, which requires that all the child trees of the twitter node are updates:

$$\mathit{twitter} \triangleq \#\mathrm{document}[\mathrm{twitter}[\Box_{\otimes}(\mathrm{true}_{\mathrm{T}} \implies \mathit{update})]] \tag{3}$$

Program reasoning: Using our assertion language, we provide a local Hoare logic for reasoning about our Featherweight DOM programs. We use the O'Hearn "fault-avoiding" interpretation of Hoare

triples: $\{P\}\mathbb{C}\{Q\}$ is semantically valid if and only if, in all states satisfying P, $\mathbb{C}$ cannot fault and, if $\mathbb{C}$ terminates, the terminal state is described by Q.

We combine standard Hoare rules with small axioms and an abstract frame rule in the style of separation logic. For example, consider the small axiom for the `getChildNodes` command:

$$\left\{ \mathrm{NAME}_{\texttt{Node}}[\mathrm{F}]_{\mathrm{FID}} \wedge \texttt{id} = \mathrm{Y} \right\}$$
$$\texttt{id = getChildNodes(Node)}$$
$$\left\{ \mathrm{NAME}_{\texttt{Node}[\mathrm{Y}/\texttt{id}]}[\mathrm{F}]_{\mathrm{FID}} \wedge \texttt{id} = \mathrm{FID} \right\}$$

The precondition has two conjuncts: the first describes a node, with an arbitrary name given by the logical variable NAME, and which is identified by the value of the `Node` expression; the second states that the current value of the `id` variable is the same as the logical variable Y. The postcondition describes the same node, but substitutes the unchanged logical variable Y into the `Node` expression (in case the expression uses the variable `id`). It also states that the `id` variable now has the value of the node's forest identifier. This axiom is small because it specifies the behavior of `getChildNodes` on only its intuitive footprint, in this case the subtree identified by `Node`.

The abstract frame rule is analogous to the separation frame rule, using separation application instead of separating conjunction, and is the key ingredient enabling us to provide abstract local reasoning about the DOM API in particular, and program libraries in general. Let $\mathbb{C}$ be the code `id = getChildNodes(node)`, where "`node`" is a program variable rather than the program expression "`Node`" used in the small axiom for `getChildNodes`. Using the rule of consequence, abstract frame and an instance of the small axiom, we can obtain the following derivation for $\mathbb{C}$:

$$\dfrac{\dfrac{\left\{ \mathrm{user}_{\texttt{node}}[\mathrm{F}]_{\mathrm{FID}} \right\} \mathbb{C} \left\{ \mathrm{user}_{\texttt{node}}[\mathrm{F}]_{\mathrm{FID}} \wedge \texttt{id} = \mathrm{FID} \right\} \text{ AXIOM}}{\left\{ \mathrm{update}[-_{\mathrm{T}} \otimes \#\mathrm{text}] \circ_{\mathrm{T}} \mathrm{user}_{\texttt{node}}[\mathrm{F}]_{\mathrm{FID}} \right\} \mathbb{C} \left\{ \mathrm{update}[-_{\mathrm{T}} \otimes \#\mathrm{text}] \circ_{\mathrm{T}} \mathrm{user}_{\texttt{node}}[\mathrm{F}]_{\mathrm{FID}} \wedge \texttt{id} = \mathrm{FID} \right\} \text{ FRAME}}}{\left\{ \mathrm{update}[\mathrm{user}_{\texttt{node}}[\mathrm{F}]_{\mathrm{FID}} \otimes \#\mathrm{text}] \right\} \mathbb{C} \left\{ \mathrm{update}[\mathrm{user}_{\texttt{node}}[\mathrm{F}]_{\mathrm{FID}} \otimes \#\mathrm{text}] \wedge \texttt{id} = \mathrm{FID} \right\} \text{ CONSEQUENCE}}$$

The use of the abstract frame rule declares that the data described by the context remains unchanged.

The `getChildNodes` command is relatively simple. The most complex command in Featherweight DOM is `appendChild(Parent, NewChild)`, which not only requires that `Parent` and `NewChild` are in the DOM tree, but also that `NewChild` is not an ancestor node of `Parent`:

$$\left\{ (\varnothing_{\mathrm{F}} \multimap (\mathrm{CG} \circ_{\mathrm{T}} (\mathrm{NAME}_{\texttt{Parent}}[\mathrm{F}]))) \circ_{\mathrm{F}} \langle \mathrm{NAME}'_{\texttt{NewChild}} \wedge \mathrm{T} \rangle_{\mathrm{F}} \right\}$$
$$\texttt{appendChild(Parent, NewChild)}$$
$$\left\{ \mathrm{CG} \circ_{\mathrm{T}} (\mathrm{NAME}_{\texttt{Parent}}[\mathrm{F} \otimes \langle \mathrm{NAME}'_{\texttt{NewChild}} \wedge \mathrm{T} \rangle_{\mathrm{F}}]) \right\}$$

The precondition describes data that can be separated into a context and a forest containing a single tree identified by `NewChild`. The context is such that, if we were to place an empty forest in the hole made by removing the subtree rooted at `NewChild`, we would still be able to extract the `Parent`. This ensures that `NewChild` cannot be an ancestor of `Parent` (if it were, there would be no way to extract the `Parent` from the context left by removing `NewChild` and its children). This specification illustrates the necessity of our contextual reasoning when specifying DOM.

Using our reasoning, we give the following simple specification of a program:

$$\left\{ \exists \mathrm{S}.\mathrm{updates}_{\texttt{node}}[\mathrm{F}_1 \otimes \mathrm{S}[\mathrm{F}_3] \otimes \mathrm{F}_2 \wedge \mathit{len}(\mathrm{F}_1) = 2] \oplus \mathrm{updatesILike}_{\texttt{publishHere}}[\varnothing_{\mathrm{F}}] \right\}$$
$$\texttt{updates = getChildNodes(node);}$$
$$\texttt{thirdKid = item(updates, 2);}$$
$$\texttt{appendChild(publishHere, thirdKid)}$$
$$\left\{ \exists \mathrm{S}.\mathrm{updates}_{\texttt{node}}[\mathrm{F}_1 \otimes \mathrm{F}_2 \wedge \mathit{len}(\mathrm{F}_1) = 2]_{\texttt{updates}} \oplus \mathrm{updatesILike}_{\texttt{publishHere}}[\mathrm{S}_{\texttt{thirdKid}}[\mathrm{F}_3]] \right\}$$

The precondition specifies that the grove contains a node labelled "updates" with at least three children and identified by `node`, and a node labelled "updatesILike" with no children and identified by `publishHere`. Our reasoning proves that, if the program is run in a state which satisfies this precondition, then it will not fault and it will only terminate in states satisfying the postcondition which specifies that the third child of `node` has been moved under `publishHere`. We can use the abstract frame rule to describe how the program behaves on larger states.

3 Reasoning about Mashups

In [1, 2], Gardner, Smith *et al.* introduced an axiomatic specification of DOM. They mainly concentrated on the DOM specification, rather than fully exploring how to reason about DOM programs. Here, we explore program reasoning about mashups, combining reasoning about DOM with reasoning about a fragment of JavaScript consisting of mashup IO commands and dynamic functions.

Defining a notion of mashup is difficult. People tend to agree that a mashup consists of a web page that combines data and code from two or more sources, although which part of this picture is the "mashup" is unclear. Two sensible definitions are that either the page doing the combination of the resources is the mashup, or that the entire set of resources being combined is the mashup. We set our terminology as follows. A *mashup component* is a single resource, associated with a URI, that provides data and code. We write a mashup component as a pair $(\mathbf{doc}, \mathbb{C})$. The data **doc** provided by a component is termed *mashup data* and code $\mathbb{C}$ is termed *mashup code*. A finite partial function $\mathbf{m}$ mapping URIs to these mashup components forms a *mashup*. A *mashup program* is a mashup where we distinguish one component within it as a starting point for program execution.

Given this terminology, our program state consists of the standard variable store and the tree-stuctured heap, together with a mashup and a function table. A *function table* $\mathbf{f}$ is a mutable mapping from function names to parameter lists and body code.

Language: We extend Featherweight DOM with two new *mashup IO* commands and with *dynamic functions*. The two new mashup IO commands are:

> `id = fetchDocument(StrExpr)`: evaluates `StrExpr` to obain a URI. If $\mathbf{m}(\text{URI}) = (\mathbf{doc}, \mathbb{C})$, it ignores $\mathbb{C}$, takes **doc**, freshens the identifiers with respect to the current heap, appends the result to the heap at grove level, and stores the new root node identifier in `id`. It faults if URI $\notin dom(\mathbf{m})$.
> `runScript(StrExpr)`: evaluates `StrExpr` as above. If $\mathbf{m}(\text{URI}) = (\mathbf{doc}, \mathbb{C})$, the command ignores **doc** and behaves as $\mathbb{C}$. It faults if URI $\notin dom(\mathbf{m})$.

Using these commands, we can begin to write the code of the Twitter mashup mentioned in the introduction. Consider the following sequence of commands with respect to a mashup with components for homepage.example.com, twitter.example.com and spamcheck.example.com:

```
document = fetchDocument("homepage.example.com");
updatesDoc = fetchDocument("twitter.example.com");
runScript("spamcheck.example.com");
```

This program fragment takes the data associated with the homepage, places it in the heap with fresh identifiers, and stores the root node identifier in `document`. It then extracts the Twitter data from the Twitter component, and runs the code associated with the spam-checker in the same environment.

To give components a way of sharing reusable parametric code, we introduce *dynamic functions*. Dynamic functions are a hybrid between procedures and the dynamic, higher-order functions used in e.g. JavaScript. Dynamic functions are named code blocks, called bodies, with an associated parameter list of arbitrary arity. They are dynamic because the binding of the function name to a body can be changed during program execution. For example, function names may be initially unassigned, have some body later in execution, and another body later still. This dynamic nature is necessary to handle both the introduction of new functions and the replacement of existing functions by mashup components when they are executed via `runScript`.

We extend Featherweight DOM and the mashup IO commands with two dynamic function statements, function introduction and function call:

> `function Name(P1, ..., Pn) { Statement; return E }`: updates the function table $\mathbf{f}$ to associate the name `Name` with the body code between the braces along with the parameter names. This command cannot fault.

`r = Name(E1, ..., En)`: extracts the body code associated with `Name` from the function table, replaces the parameters names with the associated expressions `E1` to `En`, then behaves as this code, associating the return expression of the body code with the store variable `r`. It faults if there is no function named `Name` in the table, if the passed parameter count does not match the stored function, or if the function body faults.

The only restriction on the use of dynamic functions is that function bodies may not include function introduction statements; these may only be used at the "top level" component code. Function calls, however, may appear as statements anywhere and be mutually recursive. In our example Twitter program, we will simplify notation by allowing function-call statements to be used as expressions: for example, we will use the statement `newUpdate = createTextNode(getNodeValue(updateText))`, which gets the string of the text node identified by `updateText`, and creates a new node labelled by the string. We also use the standard simplification of disallowing assignment to function parameters within a function body. To allow details of component code to remain private, we use local variables with a scope limited to a single function body and local functions scoped to a single component. This restricts local variables and functions to the lexical scope of the containing function or component. The full details can by found in the technical report[22].

Assertion language: We extend the Featherweight DOM assertion language for specifying properties of the variable store and tree-structured heap, to provide an assertion language which in addition specifies properties of a mashup and function table. The mashup assertion $\mu(\texttt{Uri})$ specifies that the mashup contains an entry for `Uri`, and the function assertion $\gamma(\texttt{Name})$ specifies that the function table contains an entry for `Name`.

Program reasoning: We extend Hoare triples to handle mashup IO commands and dynamic functions, working with judgements of the form $\mathcal{M} \vdash \{P\}\mathbb{C}\{Q\}$ where $\mathcal{M}$ is a *mashup specification* built from a collection of *component specifications*.

Component specifications enable us to use published specifications of remote components, rather than requiring the full details of their (possibly changing) data and code. In fact, the remote site will tend to prove a stronger specification than the published specification. For example, we will see that the spam-checker component of our Twitter example has the published specification that its data satisfies the assertion *true*, and the code requires a string and returns a boolean. Of course, the remote site would probably prove a more complex specification, from which the published specification can be inferred.

We define a *component specification* for a given component $(\mathbf{doc}, \mathbb{C})$ as a triple (D, C, F), where D is a *data specification* describing $\mathbf{doc}$, C is a *code specification* for $\mathbb{C}$, and F is a *function specification* for the set of functions declared in $\mathbb{C}$. A *data specification* is a Featherweight DOM assertion from section 2, in which no store variables are mentioned and all identifiers are existentially quantified. A *code specification* for code $\mathbb{C}$ is a triple (P, Q, M), where P and Q are pre- and postconditions for $\mathbb{C}$ and $M = \mathit{mods}(\mathbb{C})$ is the modification set of $\mathbb{C}$: that is, the set of all non-local names modified by the code. Just as in separation logic, the modification set is needed for a side-condition on the abstract frame rule that ensures we cannot frame away assertions about names the code will modify. Its definition is standard with three additions: function introduction is considered to modify the name of the function being introduced; the modifications of a function-call statement are the modifications of the function body; and for `runScript` statements, it is the set for the script code being run. For brevity, we write the empty code specification $(\varnothing_{\mathrm{G}}, \varnothing_{\mathrm{G}}, \{\})$ as $\varnothing_{\mathbb{C}}$. We say that two code specifications are equivalent if their pre- and postconditions are logically equivalent and their modified sets are the same. A *function specification* is a finite partial function mapping every function name introduced by a function introduction statement in $\mathbb{C}$ to a pair (C, R), where C is a code specification that describes the function body and R lists the parameters.

Recall that mashups are finite partial functions from URIs to mashup components. A *mashup specification* is a finite partial function from URIs to component specifications. We write mashup spec-

ifications as $\mathcal{M} = \{\mathrm{URI}_1 \mapsto (D_1, C_1, F_1), \ldots \mathrm{URI}_n \mapsto (D_n, C_n, F_n)\}$. Given such a mashup specification $\mathcal{M}$, the *mashup function specifications* of $\mathcal{M}$, denoted $\mathcal{F}(\mathcal{M})$, is the collection of function specifications given by $F_1, \ldots, F_n$ when they are compatible. More formally, $\mathcal{F}(\mathcal{M}) = F_1 \uplus \ldots \uplus F_n$ is defined as the natural extension of

$$(F_1 \uplus F_2)(\texttt{Name}) \triangleq \begin{cases} F_1(\texttt{Name}) \iff \texttt{Name} \in dom(F_1) \wedge \texttt{Name} \notin dom(F_2) \\ F_2(\texttt{Name}) \iff \texttt{Name} \notin dom(F_1) \wedge \texttt{Name} \in dom(F_2) \\ F_1(\texttt{Name}) \iff \begin{array}{l} F_1(\texttt{Name}) = (C_1, R) \wedge \\ F_2(\texttt{Name}) = (C_2, R) \wedge \\ C_1, C_2 \text{ equivalent} \end{array} \\ \text{otherwise undefined} \end{cases}$$

We extend Hoare triples to handle mashup IO commands and dynamic functions, working with judgements of the form $\mathcal{M} \vdash \{P\}\mathbb{C}\{Q\}$. The small axioms, standard Hoare rules and abstract frame rule are not affected by the mashup specification $\mathcal{M}$. We use $\mathcal{M}$ in the mashup IO and function rules, given here (although we ignore local names as they complicate the rule syntax without being very informative, see the technical report for full details [22]):

$$\frac{\mathcal{M}(\texttt{StrExpr})=(D,C,F)}{\mathcal{M}\vdash\{\mu(\texttt{StrExpr})\}\texttt{Id = fetchDocument(StrExpr)}\{\mu(\texttt{StrExpr})\wedge D\}}$$

$$\frac{\mathcal{M}(\texttt{StrExpr})=(D,(P,Q,M),F)}{\mathcal{M}\vdash\{\mu(\texttt{StrExpr})\wedge P\}\texttt{runScript(StrExpr)}\{\mu(\texttt{StrExpr})\wedge Q\}}$$

$$\frac{\mathcal{F}(\mathcal{M})(\texttt{Name})=((P,Q,M),(\texttt{P1},\ldots,\texttt{Pn}))\quad \mathcal{M}\vdash\{P\}\mathbb{C}\{Q\}\quad mods(\mathbb{C})=M}{\mathcal{M}\vdash\{\varnothing_{\mathrm{G}}\}\ \texttt{function Name(P1, ..., Pn)}\{\mathbb{C}\}\{\gamma(\texttt{Name})\}}$$

$$\frac{\mathcal{F}(\mathcal{M})(\texttt{Name})=((P,Q,M),(\texttt{P1},\ldots,\texttt{Pn}))}{\mathcal{M}\vdash\{\gamma(\texttt{Name})\wedge P[\texttt{Ei/Pi}]\}\ \texttt{r = Name(E1, ..., En)}\{\gamma(\texttt{Name})\wedge Q[\texttt{Ei/Pi}]\}}$$

The rule `fetchDocument` specifies that the resulting document satisfies the data specification D given by the mashup specification. The rule `runScript` specifies that the resulting code has precondition P and postcondition Q, again given by the mashup specification. Notice that we are trusting the published specification of the mashup component identified by `StrExpr`. However, when we introduce our own function, we check that its body satisfies the specification and return the assertion $\gamma(\texttt{Name})$ declaring that the function is in the function table. When calling a function, we use the pre- and postconditions given in the mashup specification, with suitable substitutions of the passed expressions for parameters.

3.1 Detailed example

We apply our mashup framework and associated reasoning to our Twitter example. This is both a simple and realistic example, illustrating the main features of a common mashup. It involves acquiring data from one remote location (the Twitter update data), and processing it using code from a second, independent remote location (the spam-checker code). The results are then consumed by the homepage. The homepage is simple, but makes use of complex remote code (the spam-checker code). This is a common pattern, as large companies such as google and yahoo provide complex services, which are used in large numbers of simple mashups. We provide simple reasoning for this Twitter example. In particular, we show that, although the spam-checker code might be complex, the published specification is simple (the function `isSpammer` requires a string and returns a boolean). For space and simplicity, this example does not fully exercise our reasoning about dynamic functions. For further examples, see the technical report [22].

Twitter Update List: The Twitter update component provides a set of recent updates as data, and has no code. Recall the twitter data "twitterdata" defined in section 2(1). We use this to define the twitter component within the mashup **m** as

$$\mathbf{m}(\text{twitter.example.com}) = (\text{twitterdata}, \texttt{skip})$$

Recall the *twitter* assertion given in section 2(3), describing data in the Twitter update format. As there is no code associated with Twitter, and hence no functions, we can compactly describe the public specification $\mathcal{M}$ for the Twitter component as

$$\mathcal{M}(\text{twitter.example.com}) = (\mathit{twitter}, \varnothing_{\mathbb{C}}, \emptyset)$$

The Spam Checking service: We have been deliberately vague about the specifics of the spam checking component, noting only that it will provide some method of filtering spam users from the list. This mirrors our concerns as developers of the homepage. We do not care how the spam-checker is implemented, so long as it provides us with a checking method in the form of an `isSpammer` function. One possible design for this spam-checker is illustrated in figure 1, and involves maintaining a database in its data, and using hidden, local functions to query that database. Many other designs are equally feasible.

$$\mathbf{m}(\text{spamcheck.example.com}) = (\mathbb{D},\ \mathbb{C}_1; \texttt{function isSpammer(name)}\ \mathbb{C}_3\ \{\mathbb{C}_2\};)$$

The public specification for the service mirrors this operational vagueness by giving very weak data specification, and by giving the `isSpammer` function a simple contract to fulfil. The implementation is free to behave however it wishes, as long as it provides a suitable function, does not alter the environment beyond adding this function, and does not access the web beyond its own data.

$$\begin{array}{l} \mathit{FSpec} \triangleq \{\texttt{isSpammer(name)} \mapsto ((\texttt{name} : \textsc{Str}, \text{ret} = \text{true} \vee \text{ret} = \text{false}, \emptyset), (\texttt{name}))\} \\ \mathcal{M}(\text{spamcheck.example.com}) = (\text{true}, \begin{pmatrix} \varnothing_{\mathrm{G}} \wedge \mu(\text{spamcheck.example.com}), \\ \varnothing_{\mathrm{G}} \wedge \mu(\text{spamcheck.example.com}) \wedge\ \gamma(\texttt{isSpammer}), \\ \{\text{isSpammer}\} \end{pmatrix}, \mathit{FSpec}) \end{array}$$

The homepage: The homepage forms the distinguished mashup component. It uses its own data, which contains an `updatesILike` node. It downloads the Twitter data and the spam-checker code, then for every update that is not spam, adds the update text as a child of the `updatesILike` node. Let $\mathbb{C}_h$ denote the code:

```
document = fetchDocument("homepage.example.com");
updatesDoc = fetchDocument("twitter.example.com");
runScript("spamcheck.example.com");
// Find the "updatesILike" HTML node, and assign its id to the variable "publishHere".
publishHere = ...                       //  (elided)
// Add each update
updates = getChildNodes(item(getChildNodes(updatesDoc), 0)); length = getLength(updates); i = 0;
while (i < length) {
  update = item(updates, 0);  i = i + 1;
  // Get the update username
  userNode = getFirstChild(update); userName = getNodeValue(userNode);
  updateText = getSecondChild(update);
  // Spam check the username
  if (isSpammer(userName) == false) {
    newUpdate = createTextNode(getNodeValue(updateText));
    appendChild(publishHere, newUpdate);
  } else {
    removeChild(updates, update);
  }
}
```

In order for this code to run without faulting, it must be associated with data that contains a (typically unique) node with name "`updatesILike`" somewhere within it. This node which will be referred to in the program using its id, which will be assigned to the variable "`publishHere`". One document which meets this requirement is

$$\mathbb{D}_h \triangleq \#\text{document}_1[\text{html}_2[\text{updatesILike}_3[\varnothing]_4]_5]$$

We can then give the component for the homepage as

$$\mathbf{m}(\text{homepage.example.com}) = (\mathbb{D}_h, \mathbb{C}_h)$$

Let $Mods$ be the set of all variables modified in $\mathbb{C}_h$. We can then specify the homepage as

$$\mathit{htmlSpec} \triangleq \quad \#\text{document}[\text{html}[\Diamond_{\text{T}\to\text{F}}\text{updatesILike}[\varnothing_{\text{F}}]]]$$

$$\mathcal{M}(\text{homepage.example.com}) = \left(\begin{pmatrix} \mathit{htmlSpec}, \\ \begin{pmatrix} \mu(\text{twitter.example.com}) \wedge \mu(\text{spamcheck.example.com}) \wedge \\ \mu(\text{homepage.example.com}) \wedge \varnothing_{\text{G}}, \\ \text{true}, \\ \mathit{Mods} \end{pmatrix} \\ \emptyset \end{pmatrix}, \right)$$

The precondition for the homepage states its dependence on the Twitter and spam-checker components and its own homepage component; other than that, it needs nothing. The postcondition is just true. This is enough to prove we have a fault-free mashup; that is, the only resources required are those given in the precondition. If, however, we were to expect our homepage component to be "mashed into" other pages, we could give a stronger postcondition that describes the homepage data, and how the only change within it is the addition of Twitter updates.

Proof of the client code: We give the proof of the homepage code with respect to $\mathcal{M}$ in figure 3, under the assumption that the other components within the mashup specification have been proven with respect to their specifications. We do not give every Hoare rule application, providing instead assertions describing the intermediary states and the loop invariant.

More daring use of dynamic functions: Since mashups exist in an unstable environment, it is sometimes sensible to write a program which can fall back on a secondary service if a primary service is unavailable. For example, in the twitter mashup above, we might have the choice between two spam-checkers: a "featurefulSpamChecker" from featurefulSpamChecker.com, which provides a superior service in the form of the function "`featureFulIsSpammer(Name)`" but may not always be available; and the "reliableSpamChecker" from reliableSpamChecker.com, which provides an inferior service in the form of the function "`reliableIsSpammer(Name)`" but is always available. Suppose further that a third party has published a mashup component which allows us to look up the maintenance schedule of the featurefulSpamChecker to determine whether it is available. This third party component provides a function called "`isOnline`" which has the specification: {true}x=isOnline(){(x $\wedge$ μ(featurefulSpamChecker.com) $\vee$ ($\neg$x)}.

Now, where the Twitter mashup above introduces the "`isSpammer`" function with the line "`runScript("spamcheck.example.com");`", we might wish instead to introduce the best available spam-checking function:

```
ℂ ≜ if (isOnline()) { runScript("featurefulSpamChecker.com");
                      function isSpammer(name) { return featureFulIsSpammer(name) }
    } else { runScript("reliableSpamChecker.com");
             function isSpammer(name) { return reliableIsSpammer(name) }
    }
```

Notice that this code may introduce either of two completely different bodies for the function "`isSpammer`". We can still reason about them however, so long as they both respect a common specification. In the

Fig. 3. The Twitter Mashup is Fault Free

case where both "featureFulIsSpammer" and "reliableIsSpammer" satisfy the specification we require for "isSpammer" (which requires a string and returns a boolean), we can prove the Hoare triple:

$$\{\mu(reliableSpamChecker.com) \wedge \gamma(isOnline)\} \quad \mathbb{C} \quad \{\gamma(isSpammer)\}$$

This demonstrates the full power of our dynamic function reasoning. Function specifications are not tied uniquely to function definitions. The inference rule for function introduction allows us to reason about any dynamically introduced function that satisfies whatever specification is required for our program.

4 Conclusions

We have presented a simple formulation of mashup programs and a reasoning system for specifying properties about such programs. With our reasoning, we can prove that mashups are fault free and provide specifications for code that are analogous to XML schema for data. We have illustrated our work with the example of a twitter mashup, which we believe is representative of many mashups found on the web. This gives hope that these techniques are not merely theoretical, but might be useful in detecting and eliminating actual software faults.

We have shown how the JavaScript code in a mashup can be given code specifications which are analogous to the XML schema for describing XML data. Each component within a mashup can be independently verified, and the resulting specifications can be published to other programmers. This provides *mashup safety*. If all components in a mashup are proven with respect to the same set of specifications, then they cannot fault. That is to say that they will never attempt to call an undefined function, access a DOM node that does not exist, attempt a prohibited DOM operation or access a web site that is not available. A side-benefit of our technique is that it results in a clear understanding of the dependancies involved in any proven mashup, without having to inspect the code of every mashup-component. This means that firewalls can be configured so that they do not accidentally block an essential component of a mashup.

This work represents a first step towards reasoning about full JavaScript. Just as Gardner, Smith *et al.* extended the reasoning of Featherweight DOM to DOM Core Level 1 [1, 2], so we hope to extend the the techniques presented in this paper to reason about the full JavaScript language, described formally by Maffeis, Mitchell and Tally in [7]. One particular challenge will be reasoning about programs which make extensive use of JavaScript's higher-order functions. While the work of Birkedal, Reus, Schwinghammer and Yang [5] promises to handle higher-order functions very generally, it is also complex. We will investigate the possibility of extending the techniques presented here for dynamic functions to provide a less powerful but more manageable reasoning method for JavaScript functions.

Notice the verbose, but relatively mechanical, nature of the proof in figure 3. All the techniques demonstrated in this paper seem enticingly automatable, and invite us to extend the work of successful separation logic tools such as jStar [23] to reasoning about DOM and JavaScript. Given suitable automation, it should be possible to dispense with the need to trust the specifications of remote components, and instead have a browser plugin check that they meet their specifications before running them. We also plan to apply our techniques to further examples and problems in the mashup area. One intriguing area is *mashup security*. For example, if your bank wanted to mash a component into their online banking system, how can they be sure the component is not surreptitiously trasmitting personal information contained within the document? One promising approach is footprint analysis [24], with which we might show that the safety footprint of an untrusted component is disjoint from the sensitive personal data.

Acknowledgements We thank EPSRC, the Royal Academy of Engineering and Microsoft Research Cambridge for their financial support.

Bibliography

[1] Gardner, P.A., Smith, G.D., Wheelhouse, M.J., Zarfaty, U.D.: Local Hoare reasoning about DOM. In: Proceedings, PODS'08. (2008)
[2] Smith, G.D.: PhD thesis. In preparation
[3] O'Hearn, P., Reynolds, J., Yang, H.: Local reasoning about programs that alter data structures. In: CSL. (2001)
[4] Calcagno, C., Gardner, P., Zarfaty, U.: Context logic and tree update. 32nd POPL'2005, ACM SIGPLAN Notices **40**(1) (2005)
[5] Schwinghammer, J., Birkedal, L., Reus, B., Yang, H.: Nested hoare triples and frame rules for higher-order store. In: CSL. (2009) 440–454
[6] Thiemann, P.: A type safe DOM API. In: Proceedings DBPL, 169–183. (2005)
[7] Maffeis, S., Mitchell, J.C., Taly, A.: An Operational Semantics for JavaScript. In: Proceedings, APLAS 2008. LNCS (2008)
[8] Hosoya, H., Pierce, B.C.: XDuce: A statically typed XML processing language. ACM Transactions on Internet Technology
[9] Thiemann, P.: Towards a type system for analyzing javascript programs. In: ESOP. (2005) 408–422
[10] Paola, C.A., Anderson, C., Giannini, P., Drossopoulou, S.: Towards type inference for javascript. In: In Proc. 19th European Conference on Object-Oriented Programming, Springer (2005) 429–452
[11] Richards, G., Lebresne, S., Burg, B., Vitek, J.: An analysis of the dynamic behavior of javascript programs. In: Proceedings PLDI '10, ACM (2010)
[12] Bloom, B., Field, J., Nystrom, N., Östlund, J., Richards, G., Strnisa, R., Vitek, J., Wrigstad, T.: Thorn: robust concurrent scripting on the jvm. In: OOPSLA Companion. (2009) 789–790
[13] AdSafe Media, Inc.: Adsafe. http://www.adsafe.org/
[14] Miller, M., M. Samuel, B.L., Awad, I., Stay, M.: Caja: Safe active content in sanitized JavaScript (2008) A Google research project.
[15] Jackson, C., Wang, H.J.: Subspace: secure cross-domain communication for web mashups (2007)
[16] C. Reis, S.G., Levey, H.: Architectural principles for safe web programs (2009)
[17] O'Hearn, P.W., Yang, H., Reynolds, J.C.: Separation and information hiding. In: POPL. (2004) 268–280
[18] Parkinson, M.J., Bierman, G.M.: Separation logic and abstraction. In: POPL '05: Proceedings of the 32nd annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, New York, NY, USA, ACM (2005) 247–258
[19] Parkinson, M.J., Bierman, G.M.: Separation logic, abstraction and inheritance. In: POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, New York, NY, USA, ACM (2008) 75–86
[20] Gardner, P., Dinsdale-Young, T., Wheelhouse, M.: Abstraction and refinement for local reasoning. In: Proceedings, VSTTE 2010. (2010)
[21] Document Object Model Level 1 Specification. http://www.w3.org/TR/REC-DOM-Level-1/
[22] Gardner, P., Smith, G., Wright, A.: Local reasoning about mashups : Technical report. (2010)
[23] Distefano, D., Parkinson, M.J.: jStar: Towards practical verification for Java. In: OOPSLA, ACM (2008)
[24] Raza, M., Gardner, P.: Footprints in local reasoning. **5**(2) (2009)
[25] Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Proceedings, LICS 2002. (2002)
[26] Gardner, P., Smith, G., Wheelhouse, M.J., Zarfaty, U.: DOM: Towards a formal specification. In: PLAN-X. (2008)
[27] Hoare: An axiomatic basis for computer programming. CACM **26** (1983)
[28] World Wide Web Consortium: HTML 4.01 Specification. (December 1999)
[29] Association, E.C.M.: ECMAScript Language Specification. (June 1997)
[30] Klose, K., Ostermann, K.: Modular logic metaprogramming. In: Proceedings OOPSLA '10, ACM (2010)

Hoare Logic for Graph Programs

Christopher M. Poskitt and Detlef Plump

Department of Computer Science
The University of York, UK

Abstract. We present a new approach for verifying programs written in GP (for Graph Programs), an experimental programming language for performing computations on graphs at a high level of abstraction. Taking a labelled graph as input, a graph program nondeterministically applies to it a number of graph transformation rules, directed by simple control constructs such as sequential composition and as-long-as-possible iteration. We adapt classical Hoare logic to the domain of graphs, and describe a system of sound proof rules for showing the partial correctness of graph programs.

1 Introduction

Rule-based graph transformation (or graph rewriting) has been studied since the 1970s, motivated by its many applications to programming and specification, and the natural visualisation that graphs and graph transformation rules give to dynamic systems (see the recent monograph [4]). Recently, graph-based programming languages have seen increased interest as a way of controlling the application of rules to graphs, in order to solve graph problems in practice. For example, in implementing a graph algorithm, a graph program might direct the application of rules to a graph such that they compute its transitive closure. In a setting where a graph represents a system state, graph programs might represent the system's operational behaviour.

Often, it is desirable to be able to prove that a graph program is correct according to some specification. Suppose that a graph program computes a colouring of a graph, encoding the colours in the labels of nodes. Can we prove that the graph program will always produce properly coloured graphs? Suppose that we model the states of an access control system with graphs, and describe the operation of logging out a user by a graph program. Can we prove that certain safety properties are conformed to by the design of the operation?

Up until now, research has tended to focus on proving the correctness of graph grammars, and sets of graph transformation rules applied arbitrarily to graphs (see, for example, [19,2,11,3,6]). A first step towards verifying graph programs was taken by Habel, Pennemann, and Rensink [7], who adopted Dijkstra's weakest preconditions approach for so-called high-level programs, which provide control constructs such as sequential composition and as-long-as-possible iteration over sets of conditional graph transformation rules. However, to the best of our knowledge, the challenge of verifying programs written in implemented

graph transformation languages — such as PROGRES [20], AGG [21], Fujaba [13] and GrGen [5] — has yet to be addressed.

In this paper, we present an approach for verifying programs in the graph programming language GP (for Graph Programs) [16,12], a nondeterministic and computationally complete language for solving problems in the domain of graphs, and for which a prototype implementation exists. Rather than adopting a weakest precondition approach, we follow Hoare's seminal paper [10] and devise a calculus of proof rules which are directed by the syntax of GP's control constructs. Similar to classical Hoare logic, our calculus aims to facilitate human-guided verification and the compositional construction of proofs.

We intend in this paper to give the reader an informal understanding of our approach, favouring intuition and examples over the full technical details. These however can be found in [18] (a preprint of which is available from the authors' websites).

The organisation of this paper is as follows. Section 2 provides a brief introduction to GP, and explores its features through an example program. Section 3 introduces E-conditions, a graph specification formalism we use in the assertions of our Hoare triples. Section 4 presents the axiom schemata and inference rules of our partial correctness proof system, and demonstrates their use in an example proof of a simple graph program. Finally, in Section 5, we conclude.

2 Graph Programs

Graph programs are constructed from two components. First, a set of conditional rule schemata; intuitively, these are graph transformation rules with variables and expressions allowed as labels. Second, a sequence of commands controlling the application of the conditional rule schemata to a provided input graph. We review conditional rule schemata and programs in turn, and discuss an example program. Technical details and further examples can be found in [16,17].

2.1 Conditional Rule Schemata

Conditional rule schemata are the "building blocks" of graph programs, each one describing a single-step transformation of a graph. Rule schemata comprise two graphs: a left graph which describes a part to be matched, and a right graph which describes what the match should be replaced with. The labels of their graphs contain expressions whose variables are instantiated in the graph matching process to integers or strings. The possible instantiations of these variables can be restricted by a rule schema condition, a simple predicate demanding particular relationships between variables, or the non-existence of edges. The expressions in labels are evaluated to integers or strings after the variables have been instantiated. Rule schemata are entirely syntactic constructs, representing possibly infinite sets of graph transformation rules (since the graphs GP operates on are labelled over an infinite label alphabet of integers and strings).

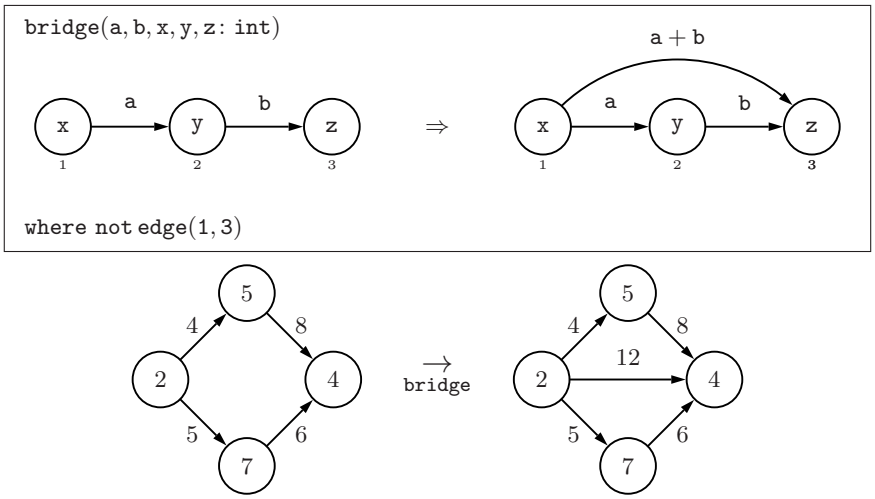


Fig. 1. A conditional rule schema and one of its applications

Figure 1 shows an example of a conditional rule schema, and a possible result of its application to a graph. The rule schema consists of the identifier `bridge` followed by the declaration of integer variables, the left and right graphs of the rule schema, the node identifiers `1`, `2`, `3` specifying which nodes are preserved, and the keyword `where` followed by a rule schema condition. Variables are instantiated with values (integers or character strings) in the graph matching process. Informally, in the application of a rule schema to a graph, a match is found for an instantiation of the left-hand side, and is replaced with the corresponding instantiation of the right-hand side. In our example, we have the following instantiation of variables: $\mathtt{x} \mapsto 2, \mathtt{y} \mapsto 5, \mathtt{z} \mapsto 4, \mathtt{a} \mapsto 4, \mathtt{b} \mapsto 8$. Observe that this is not the only possible instantiation; matches are chosen nondeterministically.

GP allows nodes and edges in rule schemata to be labelled with underscore delimited sequences, for example, `5_0` and "`York`"`_1`. Sequences can contain items of type integer and string. They are typically used to encode information into a graph, for example, to mark a node as reachable, or "tag" a node with an integer that represents its colour.

In the prototype GP programming system [12], rule schemata are constructed with a graphical editor. Labels in the left graph may only contain variables or constants (no composite expressions) because their values at execution time are determined by graph matching. The condition of a rule schema is a Boolean expression built from arithmetic expressions and the special predicate `edge`; all variables occurring in the condition must also occur in the left graph. The predicate `edge` demands the (non-)existence of an edge between two nodes in the graph to which the rule schema is applied. For example, the expression `not edge(1, 3)` in the condition of Figure 1 forbids an edge from node 1 to node

3 when the left graph is matched. The full syntax of conditions is given in [18]. Technically speaking, a rule schema is applied to a graph according to a generalisation of the double-pushout approach with relabelling [9].

2.2 Programs

We discuss an example program to familiarise the reader with GP's features. We will return to this program later in the paper to prove its correctness.

Example 1 (Colouring). A *colouring* for a graph is an assignment of colours (integers) to nodes such that the source and target of each non-looping edge have different colours. The program `colouring` in Figure 2, with the command sequence `init!; inc!`, produces a colouring for every integer-labelled input graph, recording colours as tags.

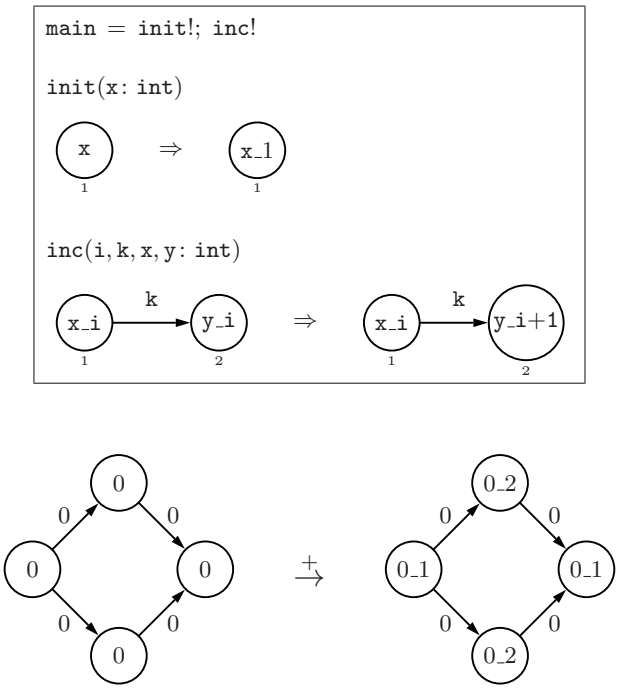


Fig. 2. The program `colouring` and one of its executions

The program initially colours each node with 1 by applying the rule schema `init` for as long as possible, using the iteration operator '`!`'. It then repeatedly increments the target colour of edges with the same colour at both ends. Note that this process is nondeterministic: Figure 2 shows an execution producing a

colouring with two colours, but a colouring with three colours could have been produced for the same input graph.

Control constructs not used in `colouring` are $\{r_1, \ldots, r_n\}$, which denotes the nondeterministic application of a rule r_i from the set, and `if` C `then` P `else` Q which executes program P if C terminates with output[1], and Q otherwise.

A full structural operational semantics is defined for GP in [17]. Each graph program is assigned a semantic function, which takes a graph as input, and returns as output the set of all graphs that could result from the execution of the program to that input graph.

3 Nested Graph Conditions with Expressions

Since the states of graph programs are graphs, and the pre- and postconditions of Hoare triples describe properties of program states, we require a specification formalism for precisely describing and reasoning about properties of graphs. The nested graph conditions of Habel and Pennemann [6] are such a specification formalism, expressively equivalent to first-order logic on graphs. Graph conditions however are unable to finitely express many properties when graphs are labelled over infinite label alphabets. For example, if we consider graphs labelled over the set of integers, it is impossible to finitely express a property as simple as "there exists an integer-labelled node"; we would require the following infinite graph condition:

$$\exists(\,\textcircled{0}\,) \vee \exists(\,\textcircled{1}\,) \vee \exists(\,\textcircled{-1}\,) \vee \exists(\,\textcircled{2}\,) \vee \exists(\,\textcircled{-2}\,) \vee \ldots$$

Since GP's label alphabet is infinite (it consists of sequences of arbitrary integers and character strings), we extend the formalism to allow expressions with variables in labels, and to have a Boolean expression restricting the instantiations of variables; we refer to what results as *E-conditions*. E-conditions are able to finitely represent infinite graph conditions. The infinite graph condition above, for example, expresses the same property as the finite E-condition $\exists(\,\textcircled{\mathtt{x}} \mid \text{type}(\mathtt{x}) = \text{int})$, where `x` is a variable that can be instantiated to any integer.

A simple example of an E-condition is $c = \exists(\,\textcircled{\mathtt{x}} \xrightarrow{\mathtt{k}} \textcircled{\mathtt{y}}\,)$, which is read "there exists at least one non-looping edge". A graph G would satisfy this E-condition, denoted $G \models c$, if variables `k`, `x`, and `y` could be instantiated to labels, which together with the nodes and edge, form a subgraph of G.

An *assignment constraint* (Boolean expression) allows one to restrict the types and values of variable instantiations. For example,

$$\exists(\,\textcircled{\mathtt{x}} \xrightarrow{\mathtt{k}} \textcircled{\mathtt{y}} \mid \text{type}(\mathtt{x}, \mathtt{y}) = \text{int} \wedge \mathtt{x} < \mathtt{y})$$

is read "there exists at least one pair of adjacent integer-labelled nodes, of which the label of the target node is larger than that of the source node".

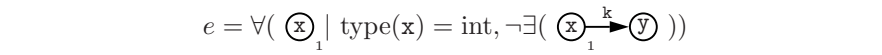
[1] Program C is tested on a *copy* of the input graph, which is subsequently discarded.

Boolean expressions over E-conditions are also E-conditions. For example,

$d = \neg\exists($  $)$

is read "there does not exist a node incident to more than one loop". Suppose that $G \models d$. Then it is the case that no instantiation of `i`, `k`, and `x` will give labels, together with a node and two loops, that form a subgraph of G.

E-conditions may also be nested. For example,

$e = \forall($  $)$

is read "no integer-labelled node has an outgoing edge to another node (with any label)". When an E-condition contains nesting, for a graph to satisfy it, we need to look further than for some simple subgraph. Suppose that $G \models e$. Then for every instantiation of `x` to an integer that, together with the single node, gives a subgraph of G, there must not be an outgoing edge from that node to any node in G with a label that `y` can be instantiated to (i.e. any node).

The formal definition of E-conditions is based on injective graph morphisms (i.e. structure preserving mappings between graphs), and allows an arbitrary amount of nesting; technical details can be found in [18].

4 A Hoare Calculus for Graph Programs

We present in this section a system of partial correctness axiom schemata and inference rules for GP, in the style of Hoare [1], using E-conditions as the assertions. We demonstrate the proof system by proving a property of our earlier `colouring` graph program.

First, we discuss what *partial correctness* means in the sense of graph programs. In the classical sense, a Hoare triple $\{s\}\ P\ \{t\}$ with s, t formulas of predicate logic and P a program fragment, is read "if P is executed when the program state satisfies s, then should P terminate, the program state will satisfy t". The execution of a graph program can follow one of three paths: it terminates with an output graph (referred to as *successful termination*), it terminates without an output graph (this is referred to as *failure*, and occurs when a rule schema, or a set of rule schemata, cannot be applied to the current graph), or it does not terminate at all (for example, `r!` will never terminate if the left graph of the rule schema is the empty graph $\emptyset$). Because of GP's nondeterminism, all three outcomes may be possible for the same program and input graph. We consider for partial correctness the successful termination case, in that if a program does terminate with an output graph, whatever that output graph may be, it satisfies some property expressed by an E-condition.

Given E-conditions c, d and a graph program P, a triple of the form $\{c\}\ P\ \{d\}$ expresses the claim that whenever a graph G satisfies c, then any graph that results from the application of P to G will satisfy d. The axiom schemata and inference rules that follow operate on such triples. As in classical Hoare logic [1], we use the proof system to construct proof trees, combining axioms and inference

rules (an example will follow). We let c, d, e, inv range over E-conditions, P, Q over arbitrary programs, r, r_i over conditional rule schemata, and $\mathcal{R}$ over sets of conditional rule schemata.

$$[\text{rule}]\ \frac{}{\{\text{Pre}(r, c)\}\ r\ \{c\}}$$

The axiom [rule] for the application of a single conditional rule schema works "backwards". Starting with a rule schema r and E-condition c as a postcondition, the transformation Pre is used to construct a precondition such that if $G \models \text{Pre}(r, c)$, and the application of r to G results in a graph H, then $H \models c$. The transformation Pre is based on graph morphisms and pushout constructions (see [18]), but informally can be described by the following steps: (1) form a disjunction of E-conditions over all possible overlappings of E-condition c and the right graph of rule schema r, (2) shift the disjunction of E-conditions from the right- to the left-hand side of r, (3) nest this within another E-condition that is universally quantified over the left graph of r.

We have that $\text{Pre}(r, c)$ implies $\text{App}(\{r\})$, where App constructs an E-condition expressing the weakest property that must be satisfied for a given rule schema set to be applicable to a graph (see below). The transformation Pre considers applicability, since otherwise, we would have to deal with failing computations.

Whereas assignment is basic to imperative programs and assignment axioms core to their correctness proofs, rule application is basic to graph programs and the [rule] axiom core to their correctness proofs.

$$[\text{ruleset}_1]\ \frac{}{\{\neg\text{App}(\mathcal{R})\}\ \mathcal{R}\ \{\text{false}\}}$$

The inference rule $[\text{ruleset}_1]$ is applied in the case that no rule schema $r \in \mathcal{R}$ can be applied to the graph. App takes as input a set $\mathcal{R}$ of conditional rule schemata, and transforms it into an E-condition describing the weakest property that a graph G must satisfy for $\mathcal{R}$ to be applicable to it. If $\mathcal{R}$ is applicable to G, then at least one rule schema $r \in \mathcal{R}$ satisfies the following: (1) it has an instantiation of variables such that its left-hand graph is isomorphic to a subgraph of G, (2) it can be applied to G without leaving dangling edges (i.e. edges which are not incident to nodes at both ends), and (3) the rule schema condition evaluates to true. The postcondition false cannot be satisfied by any graph.

$$[\text{ruleset}_2]\ \frac{\{c\}\ r_1\ \{d\}\ \ldots\ \{c\}\ r_n\ \{d\}}{\{c\}\ \{r_1, \ldots, r_n\}\ \{d\}}$$

The inference rule $[\text{ruleset}_2]$ is applied when the non-applicability of a rule schema set is not implied by the precondition. Since the rule schema to be applied is nondeterministically chosen from the set, it must be shown that the successful termination of *any* rule schema in the set results in a graph satisfying the desired postcondition, d. Note that the transformation App does not appear, since its effects are encapsulated by the transformation Pre in the axiom [rule].

$$[\text{comp})\ \frac{\{c\}\ P\ \{e\} \qquad \{e\}\ Q\ \{d\}}{\{c\}\ P; Q\ \{d\}}$$

The sequential composition rule [comp] follows its counterpart for imperative programming languages, in that we have to find an appropriate intermediate assertion, the E-condition e.

$$[\text{cons})\ c \Longrightarrow c'\ \frac{\{c'\}\ P\ \{d'\}}{\{c\}\ P\ \{d\}}\ d' \Longrightarrow d$$

Similar to its classical counterpart, the rule of consequence [cons] allows us to strengthen the precondition and weaken the postcondition (or replace them with equivalent assertions), provided that the side conditions $c \Longrightarrow c'$ and $d' \Longrightarrow d$ are valid (mechanically proving such implications of E-conditions to be valid is a problem we have not yet addressed, however, Pennemann in [14,15] has developed a resolution-like theorem prover for implications of graph conditions).

$$[\text{if})\ \frac{\{c \wedge \text{App}(\mathcal{R})\}\ P\ \{d\} \qquad \{c \wedge \neg\text{App}(\mathcal{R})\}\ Q\ \{d\}}{\{c\}\ \texttt{if}\ \mathcal{R}\ \texttt{then}\ P\ \texttt{else}\ Q\ \{d\}}$$

The conditional rule [if] formalises a case distinction based on the applicability of $\mathcal{R}$ to the input graph, utilising the transformation App.

$$[!)\ \frac{\{inv\}\ \mathcal{R}\ \{inv\}}{\{inv\}\ \mathcal{R}!\ \{inv \wedge \neg\text{App}(\mathcal{R})\}}$$

The as-long-as-possible iteration rule [!] states that if an assertion *inv* (for invariant) is satisfied after each application of $\mathcal{R}$, then once the iteration has ended, the graph will still satisfy *inv*. Additionally, since $\mathcal{R}$ is applied for as-long-as-possible, we can also deduce that $\mathcal{R}$ is no longer applicable to the graph, hence $\neg\text{App}(\mathcal{R})$ in the postcondition.

Note that two of the proof rules deal with programs that are restricted in a particular way: both the condition C of a branching command `if` C `then` P `else` Q and the body P of a loop $P!$ must be rule-set calls, that is, sets of conditional rule schemata. We gain from this restriction definability of the transformations Pre and App, but we hope to be able to modify the transformations in the future to allow arbitrary programs as input. However, despite the inconvenience of the restrictions, the computational completeness of the language is not affected, because in [8] it is shown that a graph transformation language is complete if it contains single-step application and as-long-as-possible iteration of (unconditional) sets of rules, together with sequential composition.

Example 2 (Colouring). Figure 3 is a proof tree for the `colouring` program of Figure 2. It proves that if `colouring` is executed on a graph in which the node labels are exclusively integers, then any graph resulting will have the property that each node label is an integer with a colour attached to it, and that adjacent nodes have distinct colours. That is, the proof tree proves the triple $\{\neg\exists(\ \textcircled{a}\ \mid \text{type}(\texttt{a}) \neq \text{int})\}$ `init!; inc!` $\{\forall(\ \textcircled{a}_1, \exists(\ \textcircled{a}_1 \mid \texttt{a} = \texttt{b\_c} \wedge \text{type}(\texttt{b},\texttt{c}) = \text{int})) \wedge \neg\exists(\ \textcircled{x\_i} \xrightarrow{k} \textcircled{y\_i} \mid \text{type}(\texttt{i},\texttt{k},\texttt{x},\texttt{y}) = \text{int})\}$.

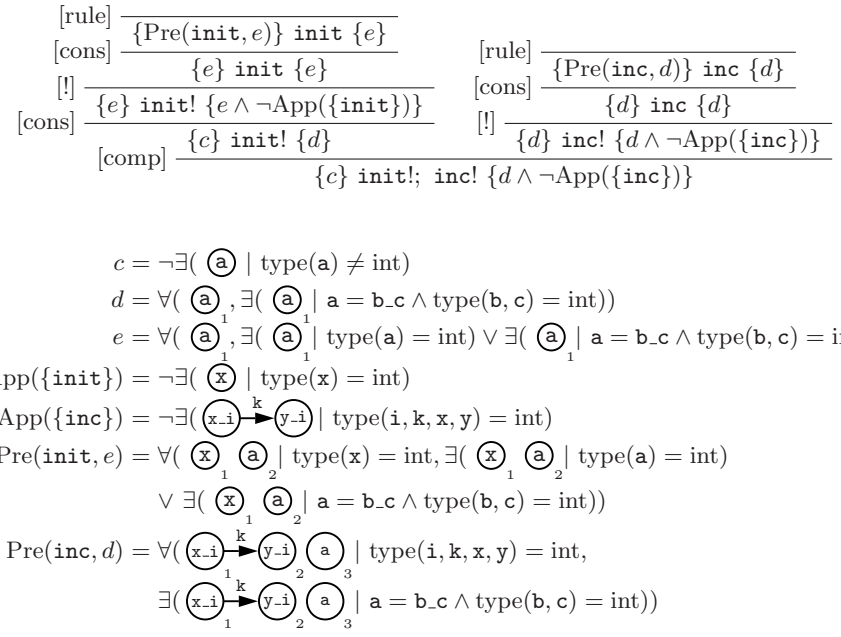


Fig. 3. A proof tree for the program `colouring` of Figure 2

The following theorem is our main technical result.

Theorem 1. *The proof system comprising* [rule], [ruleset$_1$], [ruleset$_2$], [comp], [cons], [if], *and* [!] *is sound for graph programs in the sense of partial correctness.*

This theorem is proven in [18], by showing the soundness of each of the axioms and inference rules with respect to the structural operational semantics of GP.

5 Conclusion

We have presented the first Hoare-style verification calculus for an implemented graph transformation language. This required us to extend the nested graph conditions of Habel, Pennemann, and Rensink with expressions for labels and assignment constraints, in order to deal with GP's powerful rule schemata and infinite label alphabet. We have demonstrated the use of the calculus by proving the partial correctness of a nondeterministic colouring program.

Future work will investigate the completeness of the calculus. Also, we intend to add termination proof rules in order to verify the total correctness of graph programs. Finally, we will consider how the calculus can be generalised

to deal with GP programs in which the conditions of branching statements and the bodies of loops can be arbitrary subprograms rather than just sets of rule schemata.

Acknowledgements. We are grateful to the anonymous referees for their comments which helped to improve the presentation of this paper.

References

1. Krzysztof R. Apt, Frank S. de Boer, and Ernst-Rüdiger Olderog. *Verification of Sequential and Concurrent Programs*. Springer-Verlag, third edition, 2009.
2. Paolo Baldan, Andrea Corradini, and Barbara König. A framework for the verification of infinite-state graph transformation systems. *Information and Computation*, 206(7):869–907, 2008.
3. Dénes Bisztray, Reiko Heckel, and Hartmut Ehrig. Compositional verification of architectural refactorings. In *Proc. Architecting Dependable Systems VI (WADS 2008)*, volume 5835 of *Lecture Notes in Computer Science*, pages 308–333. Springer-Verlag, 2009.
4. H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)*. Springer, 2006.
5. Rubino Geiß, Gernot Veit Batz, Daniel Grund, Sebastian Hack, and Adam M. Szalkowski. GrGen: A fast SPO-based graph rewriting tool. In *Proc. International Conference on Graph Transformation (ICGT 2006)*, volume 4178 of *Lecture Notes in Computer Science*, pages 383–397. Springer-Verlag, 2006.
6. Annegret Habel and Karl-Heinz Pennemann. Correctness of high-level transformation systems relative to nested conditions. *Mathematical Structures in Computer Science*, 19(2):245–296, 2009.
7. Annegret Habel, Karl-Heinz Pennemann, and Arend Rensink. Weakest preconditions for high-level programs. In *Proc. International Conference on Graph Transformation (ICGT 2006)*, Lecture Notes in Computer Science, pages 445–460. Springer-Verlag, 2006.
8. Annegret Habel and Detlef Plump. Computational completeness of programming languages based on graph transformation. In *Proc. Foundations of Software Science and Computation Structures (FOSSACS 2001)*, volume 2030 of *Lecture Notes in Computer Science*, pages 230–245. Springer-Verlag, 2001.
9. Annegret Habel and Detlef Plump. Relabelling in graph transformation. In *Proc. International Conference on Graph Transformation (ICGT 2002)*, volume 2505 of *Lecture Notes in Computer Science*, pages 135–147. Springer-Verlag, 2002.
10. C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.
11. Barbara König and Vitali Kozioura. Towards the verification of attributed graph transformation systems. In *Proc. Graph Transformations (ICGT 2008)*, volume 5214 of *Lecture Notes in Computer Science*, pages 305–320. Springer-Verlag, 2008.
12. Greg Manning and Detlef Plump. The GP programming system. In *Proc. Graph Transformation and Visual Modelling Techniques (GT-VMT 2008)*, volume 10 of *Electronic Communications of the EASST*, 2008.
13. Ulrich Nickel, Jörg Niere, and Albert Zündorf. The FUJABA environment. In *Proc. International Conference on Software Engineering (ICSE 2000)*, pages 742–745. ACM Press, 2000.

14. Karl-Heinz Pennemann. Resolution-like theorem proving for high-level conditions. In *Proc. Graph Transformations (ICGT 2008)*, volume 5214 of *Lecture Notes in Computer Science*, pages 289–304. Springer-Verlag, 2008.
15. Karl-Heinz Pennemann. *Development of Correct Graph Transformation Systems*. PhD thesis, Universität Oldenburg, 2009.
16. Detlef Plump. The graph programming language GP. In *Proc. Algebraic Informatics (CAI 2009)*, volume 5725 of *Lecture Notes in Computer Science*, pages 99–122. Springer-Verlag, 2009.
17. Detlef Plump and Sandra Steinert. The semantics of graph programs. In *Proc. Rule-Based Programming (RULE 2009)*, volume 21 of *Electronic Proceedings in Theoretical Computer Science*, pages 27–38, 2010.
18. Christopher M. Poskitt and Detlef Plump. A Hoare calculus for graph programs. In *Proc. International Conference on Graph Transformation (ICGT 2010)*, Lecture Notes in Computer Science. Springer-Verlag, 2010. To appear.
19. Arend Rensink, Ákos Schmidt, and Dániel Varró. Model checking graph transformations: A comparison of two approaches. In *Proc. Graph Transformations (ICGT 2004)*, volume 3256 of *Lecture Notes in Computer Science*, pages 226–241. Springer-Verlag, 2004.
20. Andy Schürr, Andreas Winter, and Albert Zündorf. The PROGRES approach: Language and environment. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2, chapter 13, pages 487–550. World Scientific, 1999.
21. Gabriele Taentzer. AGG: A graph transformation environment for modeling and validation of software. In *Applications of Graph Transformations With Industrial Relevance (AGTIVE 2003), Revised Selected and Invited Papers*, volume 3062 of *Lecture Notes in Computer Science*, pages 446–453. Springer-Verlag, 2004.

Local Reasoning about While-Loops

Thomas Tuerk

University of Cambridge Computer Laboratory
William Gates Building, JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom
http://www.cl.cam.ac.uk

Abstract. Separation logic is an extension of Hoare logic that allows local reasoning. Local reasoning is a powerful feature that often allows simpler specifications and proofs. However, this power is not used to reason about while-loops.
In this paper an inference rule is presented that allows using local reasoning to verify the partial correctness of while-loops. Instead of loop invariants this inference rule uses pre- and post-conditions for loops. This provides a different view of while-loops that is even without local reasoning often beneficial.

1 Motivation

There is a well known connection between loops and recursive procedures. Modern compilers routinely transform recursive procedures into loops as part of program optimisation and there are refactorings that depending on varying criteria transform iterative programs in recursive ones and vice-versa. However, when using separation logic, recursive implementations are often much easier to specify and verify than the corresponding imperative ones.

In the following I will look into this surprising observation a little closer considering the example of determining the length of a single-linked list. This example – like all others in this paper – has been verified using *Holfoot* [8], a *Separation logic* [5, 6] tool similar to the tool Smallfoot [1]. Both tools, Holfoot and Smallfoot, use a very similar programming language. However, Holfoot supports a richer specification language as well as interactive proofs. This enables it to reason about fully functional specifications, whereas Smallfoot just reasons about the shape of datastructures. Details about the syntax and semantics of the used specification and programming language are not important for this paper and are therefore not discussed here. The programming language is easy to understand, because its syntax is similar to C. The specification language has uncommon syntax like using underscores to denote existential quantification. Therefore, in this paper specifications will be presented in an informal, intuitive way. All specifications are just concerned with partial correctness; termination is not considered.

The examples as well as Holfoot itself are available at its webpage[1]. There is also a web-interface that allows experimenting with the examples. Holfoot

[1] http://holfoot.heap-of-problems.org

is implemented inside the HOL 4 [2, 7] theorem prover. Thus, there are formal, machine readable semantics of both the programming and the specification language and all reasoning is done by proof inside HOL 4. In particular this means that all inference rules – including the one presented in this paper – are proven correct inside HOL 4.

So, let's consider a recursive and an iterative implementation of determining the length of a single-linked list:

```
list_length(r;c) [list(c,cdata)] {
  local t;
  if (c == NULL) {
    r = 0;
  } else {
    t = c->tl;
    list_length(r;t);
    r = r + 1;
  }
} [list(c,cdata) * (r = length(cdata))]
```

```
list_length_iter(r;c) [list(c,cdata)] {
  local t;
  r = 0; t = c;
  while (t != NULL) [∃cdata1 cdata2.
     lseg(c, cdata1, t) *
     list(t, cdata2) *
     (r = length(cdata1)) *
     (cdata = cdata1 + cdata2)] {
    t = t->tl;
    r = r + 1;
  }
} [list(c,cdata) * (r = length(cdata))]
```

Both procedures get two arguments: a call-by-reference argument `r` and a call by value one `c`. The preconditions demand that `c` points to the start of a single-linked list that contains some data `cdata`. The postconditions guarantee, that this list is left unmodified and r is updated to contain the length of the list. While this specification is sufficient for the recursive implementation, the iterative one needs a complicated loop invariant. The loop-invariant (see Fig. 1)

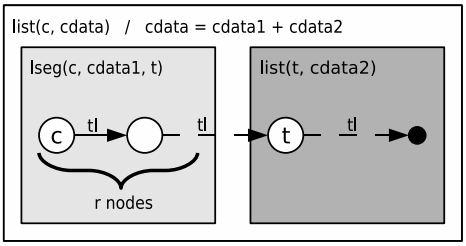


Fig. 1. Loop Invariant of list_length_iter

demands that the list can be split into two parts: the part that has already been counted and the one that still needs processing. The already counted part is a list-segment from `c` to `t` containing `cdata1`. The current value of `r` has to be the length of this already counted part. The unprocessed part is a null-terminated single-linked-list starting at `t` and containing `cdata2`. Combined, the two parts need to form the original list, i. e. appending `cdata1` and `cdata2` results in `cdata`.

Summing up, there are two implementations of the same algorithm. They have the same interface with exactly the same procedure specifications. However, while this specification is sufficient for the recursive implementation, the

iterative one needs to be annotated with a complicated loop invariant. This invariant is not just lengthy, it needs new concepts. Only the invariant needs to talk about list-segments, a partial datastructure. Both implementations do essentially the same, so why is it so much harder to specify the iterative one? The answer is that the specification of the recursive procedure call can utilise separation logic's local reasoning whereas the loop invariant does not use it.

The recursive implementation checks first[2], whether the list is empty. If it is not empty, the first element can be split off. The recursive call then determines the length of the remaining list. Thanks to local reasoning the specification of the recursive call has to mention just the tail of the original list. It is implicitly guaranteed that the first element of the original list is not modified. In contrast, the loop invariant describes the whole state / the whole datastructure. The list-segment from `c` to `t`, which is handled implicitly in the recursive implementation, is mentioned explicitly in the loop invariant.

In the following a inference rule is presented that allows the usage of local reasoning for the verification of while loops. Instead of loop invariants, this inference rule uses pre- and post-conditions for while loops. It allows to specify the list-length example as follows:

```
list_length_iter(r;c) [list(c,cdata)] {
  local t;
  r = 0; t = c;
  loop_spec [list(t,data)] {
    while (t != NULL) { t = t->tl; r = r + 1; }
  } [list(old(t),data) * (r = old(r) + length(data))]
} [list(c,cdata) * (r = length(cdata))]
```

This specification states that assuming `t` points to a start of a single linked list before the loop, then after the execution of the loop, there is still the same list in memory and the value of the variable `r` has been increased by the length of the list. So, this specification is using local reasoning. The list-segment between `c` and `t` is handled implicitly.

2 A closer look at the inference rule for while-loops

As motivated, local reasoning is an powerful feature that often allows simpler, shorter specifications and proofs. However, the inference rule for while-loops does not allow to utilise it. Let's have a closer look at this inference rule and see whether it can be modified to allow local reasoning. Hoare Logic [4] provides the following inference rule to reason about the partial correctness of while-loops:

$$\text{WHILE RULE}\quad \frac{\{c \wedge I\}\; p\; \{I\}}{\{I\}\ \textbf{while}\ \ c\ \ \textbf{do}\ \ p\ \ \textbf{done}\ \{\neg c \wedge I\}}$$

[2] Holfoot's web-interface can be used to step through the proof. It can be found at `http://holfoot.heap-of-problems.org`.

This inference rule can informally be justified by an induction on the number of loop iterations:

$$
\begin{array}{lll}
\{I\}\ \textbf{while}\ c\ \textbf{do}\ p\ \textbf{done}\ \{\neg c \wedge I\} & \Leftarrow & \text{induction / unroll} \\
\{\neg c \wedge I\}\ \{\neg c \wedge I\}\ \wedge & & \text{use induction hypothesis that} \\
\{c \wedge I\}\ p; \textbf{while}\ c\ \textbf{do}\ p\ \textbf{done}\ \{\neg c \wedge I\} & \Leftarrow & \text{the while loop satisfies} \\
 & & \text{the specification} \\
\forall \mathit{prog}.\ \{I\}\ \mathit{prog}\ \{\neg c \wedge I\} \longrightarrow & & \\
\qquad \{c \wedge I\}\ p;\ \mathit{prog}\ \ \{\neg c \wedge I\} & \Leftarrow & \text{sequential composition rule} \\
\{c \wedge I\}\ p\ \{I\} & &
\end{array}
$$

The possibility to use local reasoning is lost in the last step, the application of the composition rule. Separation logic's local reasoning guarantees that any Hoare triple can be extended by an arbitrary context R:

$$\{P\}\ \mathit{prog}\ \{Q\} \iff \forall R.\ \{P * R\}\ \mathit{prog}\ \{Q * R\}$$

However, applying the sequential composition rule in the described way, ignores the possibility to extend the specification of *prog* with a frame R. Let's try to extend the inference rule for while-loops to be able use local reasoning. This extension should be as general as possible.

Slightly generalised, the identified problem with the classical while-rule is, that it is designed for single Hoare triples $\{P\}\ \mathit{prog}\ \{Q\}$. In practice however, one often reasons about families $\{P_1\}\ \mathit{prog}\ \{Q_1\}$, $\{P_2\}\ \mathit{prog}\ \{Q_2\}$, ... of specifications. As seen with the frame rule before, such families are usually represented using higher order quantification, i. e. they are given in the form $\forall x.\ \{P(x)\}\ \mathit{prog}\ \{Q(x)\}$. If the classical while-rule is used for such a family, it results in

WHILE RULE FOR FAMILIES

$$\frac{\forall x.\ \{c \wedge I(x)\}\ p\ \{I(x)\}}{\forall x.\ \{I(x)\}\ \textbf{while}\ \ c\ \ \textbf{do}\ \ p\ \ \textbf{done}\ \{\neg c \wedge I(x)\}}$$

One can do better than this. This derived rule reasons about every member of the family, about every instantiation of x separately. The other members, other instantiations are ignored. In order to use that additional knowledge let's replay the informal justification for the classic while-rule with the quantifier in mind:

$$
\begin{array}{lll}
\forall x.\ \{I(x)\}\ \textbf{while}\ c\ \textbf{do}\ p\ \textbf{done}\ \{\neg c \wedge I(x)\} & \Leftarrow & \\
\forall x, \mathit{prog}.\ \big(\forall y.\ \{I(y)\}\ \mathit{prog}\ \{\neg c \wedge I(y)\}\big) \longrightarrow & & \\
\qquad \{c \wedge I(x)\}\ p;\ \mathit{prog}\ \ \{\neg c \wedge I(x)\} & \Leftarrow & \\
\forall x.\ \exists y.\ \{c \wedge I(x)\}\ p\ \{I(y)\}\ \wedge\ \big(\neg c \wedge I(y) \rightarrow \neg c \wedge I(x)\big) & &
\end{array}
$$

This rule is more general than the classical one. The classic one always instantiates y with x. Now the induction hypothesis is stronger and one can actually use a different instantiation for the next iteration through the loop. As discussed before, separation logic's local reasoning can be expressed as such

a family of specifications. Therefore, reasoning about families of specifications instead of single ones allows using local reasoning for loops. This is the core of the proposed inference rule for loops. It leads to the following rule:

EXTENDED WHILE RULE

$$\frac{\forall x, \mathit{prog}.\ \big(\forall y.\ \{I(y)\}\ \mathit{prog}\ \{\neg c \wedge I(y)\}\big) \longrightarrow \{c \wedge I(x)\}\ p;\ \mathit{prog}\ \ \{\neg c \wedge I(x)\}}{\forall x.\ \{I(x)\}\ \textbf{while}\ c\ \textbf{do}\ p\ \textbf{done}\ \{\neg c \wedge I(x)\}}$$

The extended while rule allows to choose for each loop iteration a different instantiation. So, I is still some kind of inductive property of the loop, but strictly speaking no invariant any more. So let's use proper pre- and postconditions for the loop. By introducing pre- and post-conditions, the case for skipping the loop becomes more complicated. There is an additional proof obligation that the pre-condition implies the post-condition. Since there is this additional proof obligation anyhow, one can easily allow code after the loop. This results in the inference rule proposed in this paper:

LOOP SPECIFICATION RULE

$$\frac{\begin{array}{c}\forall x.\ \{\neg c \wedge P(x)\}\ p_2\ \{Q(x)\} \\ \forall x, \mathit{prog}.\ \big(\forall y.\ \{P(y)\}\ \mathit{prog}\ \{Q(y)\}\big) \longrightarrow \{c \wedge P(x)\}\ p_1;\ \mathit{prog}\ \{Q(x)\}\end{array}}{\forall x.\ \{P(x)\}\ \textbf{while}\ c\ \textbf{do}\ p_1\ \textbf{done};\ p_2\ \{Q(x)\}}$$

Both extensions of the extended while rule can be justified using exactly the same reasoning as before. These extensions are natural and prove useful. Especially, using real pre- and post-conditions is convenient. However, these extensions are unrelated to the main idea of using local reasoning for loops.

Notice, that this discussion about inference rules is intentionally very informal. There has been no definition of the semantics of the programming language and no definition of Hoare triples. The purpose of this discussion is to convey the main ideas. In contrast to this informal discussion here, the implementation of this proposed inference rule in Holfoot is formal. Everything is defined using higher order logic and its soundness is machine checked using the HOL 4 theorem prover.

3 Examples

I hope, I could convince you that loop-specifications are advantageous for the initial example of calculating the length of a single-linked list. There are similar results for reversing a single-linked list, copying a single-linked list, appending two single-linked lists, removing an element from a single-linked list, etc. Due to space restrictions most of these examples are not discussed here. They can be found on Holfoot's web-page[3].

Before considering examples that are very similar to the motivating one, let's start by discussing examples that demonstrate that even without local reasoning loop specifications are still useful. In contrast to invariants, the pre- and

[3] `http://holfoot.heap-of-problems.org`

post-condition specify the behaviour of the block containing the while loop. Therefore, the loop specification rule is closely related to Eric Hehner's *specified blocks* [3]. Hehner uses single boolean expressions instead of a pre- and post-condition. Moreover, his work is much more general. However, he is not using local reasoning. Allowing for these differences, his method of reasoning about loops is very similar to the one proposed here.

3.1 Array Increment Example

Similar to Hehner's specified blocks, loop specifications slightly change how to think about loops. As a rule of thumb, loop invariants express what the loop has already done, whereas loop specifications express what it will still do. Talking about what still needs doing instead of what has already been done, often leads to more natural specifications. Even without local reasoning, Hehner prefers loops specified as blocks to invariants. He claims *that it is simpler and more direct to say what's left to be done, rather than to formulate an invariant* [3]. This difference between loop invariants and loop specifications is demonstrated by one of Hehner's examples:

```
inc(;a,n) [array(a,n,data)] {
   local i, tmp;
   i = 0;
   while (i < n) {
      tmp = (a + i) -> dta;
      (a + i) -> dta = tmp + 1;
      i = i + 1;
   }
} [array(a,n,map +1 data)]
```

This procedure increments every element of an array. The loop can be specified with the following invariant:

$$\begin{aligned}&\exists \mathit{data}_2.\ \mathrm{array}(\mathtt{a}, \mathtt{n}, \mathit{data}_2) * \\ &\qquad \big(\forall x.\quad x < \mathtt{i} \quad \Longrightarrow \mathit{data}_2[x] = \mathit{data}[x] + 1\big) * \\ &\qquad \big(\forall x.\ \mathtt{i} \le x < \mathtt{n} \Longrightarrow \mathit{data}_2[x] = \mathit{data}[x]\big)\end{aligned}$$

The invariant states that there is an array of length `n` starting at `a` and containing some existentially quantified data $data_2$. For all indices up to `i` the array contains the incremented value, for all other indices it still contains the original one. If a loop specification is used, it is the other way round:

pre: $\mathrm{array}(\mathtt{a}, \mathtt{n}, \mathit{data})$

post: $\begin{aligned}&\exists \mathit{data}_2.\ \mathrm{array}(\mathtt{a}, \mathtt{n}, \mathit{data}_2) * \\ &\qquad \big(\forall x.\quad x < \mathrm{old}(\mathtt{i}) \quad \Longrightarrow \mathit{data}_2[x] = \mathit{data}[x]\big) * \\ &\qquad \big(\forall x.\ \mathrm{old}(\mathtt{i}) \le x < \mathtt{n} \Longrightarrow \mathit{data}_2[x] = \mathit{data}[x] + 1\big)\end{aligned}$

This specification states that all the indices starting at the value of `i` will be updated, while all smaller than `i` are not touched. Notice, that no local reasoning is involved here yet. Using local reasoning, the loop specification can however be simplified by implicitly handling the part of the array that is not touched.

pre: $\mathrm{array}(\mathtt{a} + \mathtt{i}, \mathtt{n} - \mathtt{i}, \mathit{data})$

post: $\mathrm{array}(\mathtt{a} + \mathrm{old}(\mathtt{i}), \mathtt{n} - \mathrm{old}(\mathtt{i}), \mathrm{map}\ (+1)\ \mathit{data})$

This specification now states that given an array starting from $\texttt{a} + \texttt{i}$ of length $\texttt{n} - \texttt{i}$ – i. e. just the part of the original array starting at index $\texttt{i}$ – all elements of this array are incremented. There is no need any more for some complicated expressions about indices.

3.2 List Filtering Example

The last example demonstrates that loop invariants usually specify what has already been done, whereas loop specifications specify what will be done. However, both views were easy to express. The following example of filtering a list demonstrates that it might be much simpler to express what the loop will still do. Notice that this example is not exploiting local reasoning.

```
list_filter(l;x) [list(l,data)] {
  local y, z, e;
  y = l; z = NULL;
  while (y != NULL) {
     e = y->dta;
     if (e == x) { /* need to remove y */
       if (y == l) { /* first link */
          l = y->tl; dispose y; y = l;
       } else { /* not first link */
          e = y->tl; z->tl = e; dispose y; y = z->tl;
       }
     } else { /* don't need to remove y */
        z = y;  y = y->tl;
     }
  }
} [list(l, filter x from data)]
```

The loop invariant describes that parts of the list got already filtered. This partial filtering is complicated to express:

$$
\begin{array}{l}
\textbf{if } (\texttt{y} = \texttt{l}) \textbf{ then} \\
\quad \exists \mathit{data}_1.\ (\mathit{data} = (\text{some } \texttt{xs}) + \mathit{data}_1) * \text{list}(\texttt{l}, \mathit{data}_1) \\
\textbf{else} \\
\quad \exists \mathit{data}_1, \mathit{date}, \mathit{data}_2.\ (\mathit{data} = \mathit{data}_1 + \mathit{date} + (\text{some } \texttt{xs}) + \mathit{data}_2) * \\
\qquad\qquad\qquad\qquad \text{lseg}(\texttt{l}, \text{filtered } \mathit{data}_1, \texttt{z}) * (z \mapsto [\texttt{tl} : \texttt{y},\ \texttt{dta} : \mathit{date}]) * \\
\qquad\qquad\qquad\qquad \text{lseg}(\texttt{y}, \mathit{data}_2)
\end{array}
$$

This invariant is even worse than it looks, because the shorthand (some $\texttt{x}s$) is used to denote a list of unknown length that consists of just the element $\texttt{x}$. In contrast, the loop specification is straightforward, because it describes that the whole list starting at $\texttt{y}$ will be filtered.

$$
\begin{array}{ll}
\textbf{pre:} & \text{list}(\texttt{y}, \mathit{data}_2) * \\
& \textbf{if } (\texttt{y} \neq \texttt{l}) \textbf{ then } \text{lseg}(\texttt{l}, \mathit{data}, \texttt{z}) * (\texttt{z} \mapsto [\texttt{tl} : \texttt{y}, \texttt{dta} : z\mathit{date}]) \textbf{ else } \text{emp} \\
\textbf{post:} & \textbf{if } (\text{old}(\texttt{y}) = \text{old}(\texttt{l})) \textbf{ then } \text{list}(\texttt{l}, \text{filtered } \mathit{data}_2) \\
& \textbf{else } \text{list}(\texttt{l}, \mathit{data} + z\mathit{date} + \text{filtered } \mathit{data}_2)
\end{array}
$$

3.3 List Copy Example

After considering examples for which loop specifications proved beneficial even without local reasoning, let's have a look at an example with local reasoning:

```
list_copy(z;c) [list(c,data)] {
  local x,y,w,d;
  if (c==NULL) { z=NULL; }
  else {
    z=new(); z->tl=NULL; x = c->dta; z->dta = x; w=z; y=c->tl;
    while (y!=NULL) {
      d=new(); d->tl=NULL; x=y->dta; d->dta=x; w->tl=d; w=d; y=y->tl;
    }
  }
} [list(c,data) * list(z,data)]
```

This procedure copies a single-linked list that starts at $\mathtt{c}$ and updates the call-by-reference argument $\mathtt{z}$ such that $\mathtt{z}$ points to the copy after execution. The procedure first checks, whether the list is empty. In this case, nothing needs to be copied. Otherwise, the first element is copied and auxiliary variables $\mathtt{w}$ and $\mathtt{y}$ initialised. After this initialisation, $\mathtt{z}$ points to the beginning of the copy, $\mathtt{w}$ points to its last element and $\mathtt{y}$ points to the part of the original list that still needs to be copied. Then a while loop is used to copy the remainder of the list by copying the element pointed to by $\mathtt{y}$ and then advancing $\mathtt{y}$ and $\mathtt{w}$.

The while-loop can be specified with the following invariant:

$$\begin{aligned}\exists \mathit{data}_1,\ \mathit{cdate},\ \mathit{data}_2.\ &(\mathit{data} = \mathit{data}_1 + \mathit{cdate} + \mathit{data}_2) * \\ &\mathrm{lseg}(\mathtt{c}, \mathit{data}_1 + \mathit{cdate}, y) * \\ &\mathrm{lseg}(\mathtt{z}, \mathit{data}_1, w)\ *\ (\mathtt{w} \mapsto [\mathtt{tl} : 0, \mathtt{dta} : \mathit{cdate}]) * \\ &\mathrm{list}(\mathtt{y}, \mathit{data}_2)\end{aligned}$$

This invariant states that the original data can be split into three parts: two lists data_1, data_2 and a single element *cdate*. There is a list-segment from $\mathtt{c}$ to $\mathtt{y}$ containing data_1 followed by *cdate*. This part of the original list has already been copied. The data data_1 has been copied to a list-segment from $\mathtt{z}$ to $\mathtt{w}$. The last entry *cdate* is pointed to by $\mathtt{w}$. Finally, data_2 still needs to be copied. It is stored in a list starting at $\mathtt{y}$.

Using a loop specification simplifies reasoning about the loop significantly:

$$\begin{aligned}&\textbf{pre:}\quad \mathtt{w} \mapsto [\mathtt{tl} : 0, \mathtt{dta} : \mathit{cdate}]\ *\ \mathrm{list}(\mathtt{y}, \mathit{data}_2) \\ &\textbf{post:}\quad \mathrm{list}(\mathrm{old}(\mathtt{w}), \mathit{cdate} + \mathit{data}_2)\ *\ \mathrm{list}(\mathrm{old}(\mathtt{y}), \mathit{data}_2)\end{aligned}$$

This specification states that if before the loop is executed $\mathtt{w}$ points to some data *cdate* and there is a list starting at $\mathtt{y}$ containing $\mathtt{data}_2$, then the list starting at $\mathtt{y}$ is copied such that the old value of $\mathtt{w}$ points to a list containing *cdate* followed by data_2 after the execution of the loop. The part of the list that has already been copied, i. e. the list-segment from $\mathtt{c}$ to $\mathtt{y}$ does not need to be mentioned explicitly. It is handled implicitly using local reasoning. Notice moreover that the loop specification does not use list-segments.

3.4 Partial Datastructures

Loop specifications can utilise local reasoning in order to implicitly handle some part of the state that loop invariants mention explicitly. This implicitly handled part of the state is usually a partial datastructure. For the examples so far, these partial data structures are easy to express. For lists, the partial datastructure is a list-segment and for arrays it is an array. Let's now consider a slightly more complicated datastructure: trees. For trees, the corresponding partial datastructure is a tree with a hole for some other tree. This is difficult to express. Separation logic's magic-wand operator can be used, but reasoning about this additional operator is not straightforward and Holfoot is not able to do it. Therefore, Holfoot usually can't handle the invariants of loops that operate on trees. However, loop specifications can be used to avoid the partial datastructure. This allows Holfoot to reason about additional examples like the following:

```
search_tree_delete_min (t,m;) [binary_search_tree(t;keys) * (keys ≠ ∅)] {
  local tt, pp, p;
  p = t->l;
  if (p == 0) { m = t->dta; tt = t->r; dispose (t); t = tt; } else {
    pp = t; tt = p->l;
    loop_spec [binary_search_tree(p, keys2) & (pp points to p and p to tt)] {
      while (tt != NULL) { pp = p; p = tt; tt = p->l; }
      m = p->dta; tt = p->r; dispose(p); pp->l = tt;
    } [∃p2. binary_search_tree(p2, keys2 without min(keys2)) & (pp points to p2)]
  }
} [binary_search_tree(t;keys without min(keys)) * (m = min(keys))]
```

This procedure deletes the minimal key from a non-empty binary search tree. The while-loop is used to search for the node storing the minimal key. After the loop has been executed, the original binary-search tree is unmodified and the variable `p` points to the node holding the minimal key and `pp` to its parent node. However, expressing these properties of `p` and `pp` is complicated and would require some kind of partial tree datastructure. Therefore, the code that deletes the minimal element is included in the loop specification. Thus, the post-condition of the loop specification can state, that the minimal key of the original tree has been deleted. In contrast to the corresponding loop invariant, the loop specification does not need partial tree datastructures.

Besides demonstrating that loop specifications can be used to eliminate the need for partial datastructures, the last example also demonstrates why it is useful that loop specifications allow code after the while-loop. Allowing code after the loop is a minor extension, that is not used by most of the examples that I considered so far. However, as this example illustrates, it sometimes results in much simpler post-conditions.

4 Conclusion

In this paper an additional inference rule for while loops is presented. This *loop specification rule* uses pre- and post-conditions instead of invariants. Loop invariants express what the loop has done so far. In contrast loop specifications state what the loop will still do. This often leads to more natural specifications.

The loop specifications presented here are very similar to Eric Hehner's specified blocks [3]. Even without local reasoning they often lead to simpler, more natural specifications as demonstrated by the list filtering example. However, they have mainly been introduced in order to be able to use separation logic's local reasoning for loops. Using local reasoning, loop specifications gain their full potential. Besides leading to even simpler specification, local reasoning can be used to avoid the need for predicates describing partial datastructures.

Loop specifications have been implemented inside Holfoot. This implementation includes a formal correctness proof inside the HOL 4 theorem prover. There are many Holfoot examples available that demonstrate that loop specifications can simplify the specification and verification of loops considerably. There are examples for single linked lists like reversing a single-linked list, copying a single-linked list, appending two single-linked lists, removing an element from a single-linked list, examples for arrays like copying an array, binary search, quicksort and examples for binary trees like binary search tree lookup and deletion or traversing a tree with a user managed stack. These examples and many others can be found on Holfoot's webpage [4]. The binary tree examples might be especially interesting. These could not be handled by Holfoot without loop specifications, because Holfoot does not support predicates that are able to describe the otherwise necessary partial datastructures.

Acknowledgements

I would like to thank Rustan Leino, David Naumann, Peter O'Hearn, Matthew Parkinson and Hongseok Yang for discussions, comments and especially for pointing me to Eric Hehner's work.

[4] `http://holfoot.heap-of-problems.org`

Bibliography

[1] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *FMCO*, volume 4111 of *Lecture Notes in Computer Science*, pages 115–137. Springer, 2005.

[2] M.J.C. Gordon and T.F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University, 1993.

[3] Eric C. R. Hehner. Specified blocks. In Bertrand Meyer and Jim Woodcock, editors, *VSTTE*, volume 4171 of *Lecture Notes in Computer Science*, pages 384–391. Springer, 2005.

[4] C. A. R. Hoare. An axiomatic basis for computer programming. *COMMUNICATIONS OF THE ACM*, 12(10):576–580, 1969.

[5] P.W. O'Hearn, J.C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Proceedings of 15th Annual Conference of the European Association for Computer Science Logic*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer-Verlag, September 2001.

[6] J.C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS '02: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society.

[7] Konrad Slind and Michael Norrish. A brief overview of HOL4. In Otmane Aït Mohamed, César Muñoz, and Sofiène Tahar, editors, *TPHOLs*, volume 5170 of *Lecture Notes in Computer Science*, pages 28–32. Springer, 2008.

[8] Thomas Tuerk. A formalisation of Smallfoot in HOL. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *TPHOLs*, volume 5674 of *Lecture Notes in Computer Science*, pages 469–484. Springer, 2009.

Loop Invariant Synthesis in a Combined Domain (Extended Abstract)

Chenguang Luo[1], Guanhua He[1], Shengchao Qin[2], and Wei-Ngan Chin[3]

[1] Durham University, Durham, DH1 3LE, UK
[2] Teesside University, Middlesbrough, TS1 3BA, UK
[3] National University of Singapore
{chenguang.luo,guanhua.he}@dur.ac.uk,s.qin@tees.ac.uk
chinwn@comp.nus.edu.sg

Abstract. Automated verification of memory safety and functional correctness for heap-manipulating programs has been a challenging task, especially when dealing with complex data structures with strong invariants involving both shape and numerical properties. Existing verification systems usually rely on users to supply annotations, which can be tedious and error-prone, and can significantly restrict the scalability of the verification system. In this paper, we reduce the need of user annotations by automatically inferring loop invariants over an abstract domain with both separation and numerical information. Our loop invariant synthesis is conducted automatically by a fixpoint iteration process, equipped with newly designed abstraction mechanism, and join and widening operators. Initial experiments have confirmed that we can synthesise loop invariants with non-trivial constraints.

1 Introduction

Although it is a challenging problem to automatically verify heap manipulating programs written in mainstream imperative languages, dramatic advances have been made since the emergence of separation logic [8, 13]. Examples include the Smallfoot tool [1] for the verification on pointer safety, and Hip/Sleek [10, 11] for more general properties (both structural and numerical ones).

However, one problem of these verification systems is that they generally require users to provide specifications for each method as well as invariants for each loop, which is both tedious and error-prone. This also affects their scalability, as there can be many methods in a program and each method may still contain several while loops.

To conquer this problem, separation logic based shape analysis techniques are brought in, e.g., the SpaceInvader tool [2, 5, 15]. As a further step of Smallfoot, it automatically infers method specifications and loop invariants for pointer safety in the shape domain. Other works such as THOR [9] incorporate simple numerical information into the shape domain to allow automated synthesis of properties like list length. Their success proves the necessity and feasibility for shape analysis to help automate the verification process.

However, the prior analyses focus mainly on relatively simple properties, such as pointer safety for lists and list length information. It is difficult to apply them in the presence of more sophisticated program properties, such as:

- More flexible user-defined data structures, such as trees;
- Relational numerical properties, like sortedness and binary search property.

These properties can be part of the full functional correctness of heap-manipulating programs. The (aforementioned) Hip/Sleek tool aims to verify such properties and it allows users to define their own shape predicates to express their desired level of correctness.

In this paper, we make the first stride to improve the level of automation for Hip/Sleek-like verification systems by discovering loop invariants automatically over the combined shape and numerical domain. This proves to be a challenging problem especially since we aim towards full functional correctness that Hip/Sleek targets at. Our approach is based on the framework of abstract interpretation [4] with fixpoint computation.

In summary, this paper makes the following contributions:

- We propose a loop invariant synthesis with novel operations for abstraction, join and widening over a combined shape and numerical domain.
- We demonstrate that our analysis is sound and terminates.
- We have integrated our solution with Hip/Sleek and conducted some initial experiments. The experimental results confirm the viability of our solution.

We shall next illustrate our approach informally via an example before presenting the formal details of abstraction, join and widening.

2 The Approach

Separation logic [8, 13] extends Hoare logic to support reasoning about shared mutable data structures. It provides separation conjunction ($*$) to form formulae like $\mathtt{p}_1 * \mathtt{p}_2$ to assert that two heaps described by $\mathtt{p}_1$ and $\mathtt{p}_2$ are domain-disjoint. Our abstract domain is founded on a hybrid logic of both separation logic and classical first-order logic to specify both separation and numerical properties. Similar to the Hip/Sleek system, we allow user-defined inductive predicates. For example, with a data structure definition for a node in a list `data node { int val; node next; }`, we can define a predicate for a list as

$$\mathtt{root}{::}\mathtt{ll}\langle \mathtt{n}\rangle \equiv (\mathtt{root}{=}\mathtt{null} \wedge \mathtt{n}{=}0) \vee (\exists \mathtt{v}, \mathtt{q}, \mathtt{m} \cdot \mathtt{root}{::}\mathtt{node}\langle \mathtt{v}, \mathtt{q}\rangle * \mathtt{q}{::}\mathtt{ll}\langle \mathtt{m}\rangle \wedge \mathtt{n}{=}\mathtt{m}{+}1)$$

The parameter `root` for the predicate `ll` is the root pointer referring to the list. Its length is denoted by `n`. A uniform notation $\mathtt{p}{::}\mathtt{c}\langle \mathtt{v}^*\rangle$ is used for either a singleton heap or a predicate. If `c` is a data node, the notation represents a singleton heap, $\mathtt{p} \mapsto \mathtt{c}[\mathtt{v}^*]$, e.g. the $\mathtt{root}{::}\mathtt{node}\langle \mathtt{v}, \mathtt{q}\rangle$ above. If `c` is a predicate name, then the data structure pointed to by `p` has the shape `c` with parameters $\mathtt{v}^*$, e.g., the $\mathtt{q}{::}\mathtt{ll}\langle \mathtt{m}\rangle$ above.

We can also define a list segment as follows:

$$\mathtt{ls}\langle \mathtt{p}, \mathtt{n}\rangle \equiv (\mathtt{root}{=}\mathtt{p} \wedge \mathtt{n}{=}0) \vee (\mathtt{root}{::}\mathtt{node}\langle \_, \mathtt{q}\rangle * \mathtt{q}{::}\mathtt{ls}\langle \mathtt{p}, \mathtt{m}\rangle \wedge \mathtt{n}{=}\mathtt{m}{+}1)$$

where we use the following shortened notation: (i) default `root` parameter in LHS may be omitted, (ii) unbound variables, such as `q` and `m`, are implicitly

existentially quantified, and (iii) the underscore _ denotes an existentially quantified anonymous variable.

If users want to verify a sorting algorithm, they can incorporate sortedness property into the above predicates as follows:

$$\texttt{sll}\langle \texttt{n}, \texttt{mn}, \texttt{mx}\rangle \equiv (\texttt{root}{=}\texttt{null} \wedge \texttt{n}{=}0 \wedge \texttt{mn}{=}\texttt{mx}) \vee (\texttt{root}{::}\texttt{node}\langle \texttt{mn}, \texttt{q}\rangle * \texttt{q}{::}\texttt{sll}\langle \texttt{n}_1, \texttt{k}, \texttt{mx}\rangle \wedge \texttt{mn}{\leq}\texttt{k} \wedge \texttt{n}{=}\texttt{n}_1{+}1)$$

$$\texttt{sls}\langle \texttt{p}, \texttt{n}, \texttt{mn}, \texttt{mx}\rangle \equiv (\texttt{root}{=}\texttt{p} \wedge \texttt{n}{=}0 \wedge \texttt{mn}{=}\texttt{mx}) \vee (\texttt{root}{::}\texttt{node}\langle \texttt{mn}, \texttt{q}\rangle * \texttt{q}{::}\texttt{sls}\langle \texttt{p}, \texttt{n}_1, \texttt{k}, \texttt{mx}\rangle \wedge \texttt{mn}{\leq}\texttt{k} \wedge \texttt{n}{=}\texttt{n}_1{+}1)$$

where mn and mx denote resp. the min and max values stored in the sorted list.

Such user-supplied predicates can be used to specify loop invariants and method pre/post-specifications. HIP/SLEEK also allows user-defined lemmas to express coercion relations between predicates [10]. For example, we can express with lemma that a list is formed by a list segment with another list as

$$\texttt{root}{::}\texttt{ll}\langle \texttt{n}\rangle \wedge \texttt{n}{=}\texttt{n}_1{+}\texttt{n}_2 \longleftarrow \texttt{root}{::}\texttt{ls}\langle \texttt{p}, \texttt{n}_1\rangle * \texttt{p}{::}\texttt{ll}\langle \texttt{n}_2\rangle$$

This lemma mechanism is also supported by our analysis. It can benefit our abstraction operation significantly (described later).

We now illustrate via an example our loop invariant synthesis process.

```
0 data node { int val;                9   while (srt != null &&
              node next; }                          srt.val <= v) {
1 node ins_sort(node x)              10     prv=srt; srt=srt.next;
2  requires x::ll⟨n⟩                 11   }
3  ensures res::sll⟨n,mn,mx⟩         12   cur.next=srt;
4 {int v;                            13   if (prv != null) prv.next=cur;
5  node res,cur,srt,prv=null;        14   else res=cur;
6  while (x != null) {               15  }
7   cur=x; x=x.next; v=cur.val;      16  return res;
8   srt=res; prv=null;               17 }
```

Fig. 1. Insertion sort for linked list.

The method ins_sort (Figure 1) sorts a linked list with the insertion sort algorithm. It is implemented with two nested while loops. The outer loop traverses the whole list x, takes out each node from it (line 7), and inserts that node into another already sorted list r (which is empty initially before the sorting). This insertion process makes use of the inner while loop in lines 9-11 to look for a proper position in the already sorted list for the new node to be inserted. The actual insertion takes place at lines 12-14.

To verify this program, we need to synthesise appropriate loop invariants for both while loops. Our analysis follows a standard fixpoint iteration process. It starts with the (abstract) program state immediately before the while loop (i.e., the initial state) and symbolically execute the loop body for several iterations, until the obtained states converge to a fixpoint, which is the loop invariant. During such process, at the end of each iteration, we use an abstraction function to eliminate existentially quantified logical variables, and the join and widening operators to guarantee termination for the numerical domain.

As for our example, due to the presence of nested loops, each iteration of the analysis for the outer loop actually infers a loop invariant for the inner loop. We shall now illustrate how we synthesise a loop invariant for the inner loop.

Suppose that in one iteration for the outer loop, the state at line 9 becomes

$$\texttt{res}::\texttt{sll}\langle \texttt{n}_\texttt{r}, \texttt{a}, \texttt{b}\rangle * \texttt{cur}::\texttt{node}\langle \texttt{v}, \texttt{x}\rangle * \texttt{x}::\texttt{ll}\langle \texttt{n}_\texttt{x}\rangle \wedge \texttt{srt}{=}\texttt{res} \wedge \texttt{prv}{=}\texttt{null} \wedge \texttt{n}_\texttt{r}{+}\texttt{n}_\texttt{x}{+}1{=}\texttt{n}$$

Note that since the inner loop does not mutate the heap part referred to by cur and x (i.e., $\texttt{cur}::\texttt{node}\langle \texttt{v}, \texttt{x}\rangle * \texttt{x}::\texttt{ll}\langle \texttt{n}_\texttt{x}\rangle$), we can ignore it during the invariant synthesis and add it back to the program state using the frame rule of separation logic [13]. Therefore, the initial state for loop invariant synthesis becomes

$$\texttt{res}::\texttt{sll}\langle \texttt{n}_\texttt{r}, \texttt{a}, \texttt{b}\rangle \wedge \texttt{srt}{=}\texttt{res} \wedge \texttt{prv}{=}\texttt{null} \wedge \texttt{n}_\texttt{r}{+}\texttt{n}_\texttt{x}{+}1{=}\texttt{n} \tag{1}$$

From this state, symbolically executing the loop body once yields the state:

$$\begin{array}{c}\texttt{res}::\texttt{node}\langle \texttt{a}, \texttt{srt}\rangle * \texttt{srt}::\texttt{sll}\langle \texttt{n}_\texttt{s}, \texttt{c}_1, \texttt{b}\rangle \wedge \texttt{prv}{=}\texttt{res} \wedge \\ \texttt{a}{\le}\texttt{c}_1 \wedge \texttt{a}{\le}\texttt{v} \wedge \texttt{n}_\texttt{r}{+}1{=}\texttt{n}{-}\texttt{n}_\texttt{x} \wedge \texttt{n}_\texttt{s}{+}1{=}\texttt{n}_\texttt{r}\end{array} \tag{2}$$

which says that pointer srt moves towards the tail of the list for one node. Then we join it with the initial state (1) yielding

$$\begin{array}{l}(\texttt{res}::\texttt{sll}\langle \texttt{n}_\texttt{r}, \texttt{a}, \texttt{b}\rangle \wedge \texttt{srt}{=}\texttt{res} \wedge \texttt{prv}{=}\texttt{null} \wedge \texttt{n}_\texttt{r}{+}\texttt{n}_\texttt{x}{+}1{=}\texttt{n}) \vee \\ (\texttt{res}::\texttt{node}\langle \texttt{a}, \texttt{srt}\rangle * \texttt{srt}::\texttt{sll}\langle \texttt{n}_\texttt{s}, \texttt{c}_1, \texttt{b}\rangle \wedge \\ \qquad \texttt{prv}{=}\texttt{res} \wedge \texttt{a}{\le}\texttt{c}_1 \wedge \texttt{a}{\le}\texttt{v} \wedge \texttt{n}_\texttt{r}{+}1{=}\texttt{n}{-}\texttt{n}_\texttt{x} \wedge \texttt{n}_\texttt{s}{+}1{=}\texttt{n}_\texttt{r})\end{array} \tag{3}$$

The second iteration over the loop body starts with (3) and exhibits (also) the case that srt runs two nodes towards tail, while prv goes one node. Its result is then joined with pre-state (1) to become the current state:

$$\begin{array}{c}(3) \vee \texttt{res}::\texttt{node}\langle \texttt{a}, \texttt{prv}\rangle * \texttt{prv}::\texttt{node}\langle \texttt{c}_1, \texttt{srt}\rangle * \texttt{srt}::\texttt{sll}\langle \texttt{n}_\texttt{s}, \texttt{c}_2, \texttt{b}\rangle \wedge \\ \texttt{a}{\le}\texttt{c}_1{\le}\texttt{c}_2 \wedge \texttt{c}_1{\le}\texttt{v} \wedge \texttt{n}_\texttt{r}{+}1{=}\texttt{n}{-}\texttt{n}_\texttt{x} \wedge \texttt{n}_\texttt{s}{+}2{=}\texttt{n}_\texttt{r}\end{array} \tag{4}$$

Executing the loop body a third time returns a post-state where three nodes are passed by srt, and two by prv, as below:

$$\begin{array}{c}(4) \vee \texttt{res}::\texttt{node}\langle \texttt{a}, \texttt{r}_0\rangle * \texttt{r}_0::\texttt{node}\langle \texttt{c}_1, \texttt{prv}\rangle * \texttt{prv}::\texttt{node}\langle \texttt{c}_2, \texttt{srt}\rangle * \\ \texttt{srt}::\texttt{sll}\langle \texttt{n}_\texttt{s}, \texttt{c}_3, \texttt{b}\rangle \wedge \texttt{a}{\le}\texttt{c}_1{\le}\texttt{c}_2{\le}\texttt{c}_3 \wedge \texttt{c}_2{\le}\texttt{v} \wedge \texttt{n}_\texttt{r}{+}1{=}\texttt{n}{-}\texttt{n}_\texttt{x} \wedge \texttt{n}_\texttt{s}{+}3{=}\texttt{n}_\texttt{r}\end{array}$$

where we have an auxiliary logical variable $\texttt{r}_0$. Following this trend, it is predictable that every iteration hereafter will introduce an additional logical variable (referring to a list node). If we indulge in such increase in the subsequent iterations, the analysis will never terminate. Our abstraction process prevents this from happening by eliminating such logical variables as follows:

$$\begin{array}{c}(4) \vee \texttt{res}::\texttt{sls}\langle \texttt{prv}, \texttt{n}_1, \texttt{a}, \texttt{c}_1\rangle * \texttt{prv}::\texttt{node}\langle \texttt{c}_2, \texttt{srt}\rangle * \texttt{srt}::\texttt{sll}\langle \texttt{n}_\texttt{s}, \texttt{c}_3, \texttt{b}\rangle \wedge \\ \texttt{a}{\le}\texttt{c}_1{\le}\texttt{c}_2{\le}\texttt{c}_3 \wedge \texttt{c}_2{\le}\texttt{v} \wedge \texttt{n}_\texttt{r}{+}1{=}\texttt{n}{-}\texttt{n}_\texttt{x} \wedge \texttt{n}_\texttt{s}{+}3{=}\texttt{n}_\texttt{r} \wedge \texttt{n}_1{=}2\end{array}$$

Note that the heap part $\texttt{res}::\texttt{node}\langle \texttt{a}, \texttt{r}_0\rangle * \texttt{r}_0::\texttt{node}\langle \texttt{c}_1, \texttt{prv}\rangle$ is abstracted as a sorted list segment $\texttt{res}::\texttt{sls}\langle \texttt{prv}, \texttt{n}_1, \texttt{a}, \texttt{c}_1\rangle$ with $\texttt{n}_1$ denoting the length of the segment and $\texttt{n}_1{=}2$ added into the state. This abstraction process ensures that our analysis does not allow the shape to increase infinitely.

Executing the loop body a fourth time returns a post-state where four nodes are passed by srt, and three by prv. Therefore an abstraction is performed to remove the logical pointer variables. To simplify presentation, we denote σ as $\texttt{res}::\texttt{sls}\langle \texttt{prv},\texttt{n}_1,\texttt{a},\texttt{c}_1\rangle * \texttt{prv}::\texttt{node}\langle \texttt{c}_2,\texttt{srt}\rangle * \texttt{srt}::\texttt{sll}\langle \texttt{n}_\texttt{s},\texttt{c}_3,\texttt{b}\rangle \wedge \texttt{a}{\le}\texttt{c}_1{\le}\texttt{c}_2{\le}\texttt{c}_3 \wedge \texttt{c}_2{\le}\texttt{v} \wedge \texttt{n}_\texttt{r}{+}1{=}\texttt{n}{-}\texttt{n}_\texttt{x}$, and the abstracted result (after the fourth iteration) is

$$(4) \vee (\sigma \wedge \texttt{n}_\texttt{s}{+}3{=}\texttt{n}_\texttt{r} \wedge \texttt{n}_1{=}2) \vee (\sigma \wedge \texttt{n}_\texttt{s}{+}4{=}\texttt{n}_\texttt{r} \wedge \texttt{n}_1{=}3)$$

for which we have an observation that the last two disjunctions share the same shape part (as in σ). Therefore, when joined with the previous state, the disjunction will be transferred to the numerical domain, as follows:

$$(4) \vee (\sigma \wedge (\mathtt{n_s}{+}3{=}\mathtt{n_r} \wedge \mathtt{n_1}{=}2 \vee \mathtt{n_s}{+}4{=}\mathtt{n_r} \wedge \mathtt{n_1}{=}3))$$

This simplifies the abstraction further. After that, our widening operation compares the current state with the previous one, to look for the same (numerical) constraints that both states imply, and to replace those numerical constraints in the current state with the ones discovered by widening. This operation eventually ensures termination of our analysis. As for the example, some constraints among $\mathtt{n_s}$, $\mathtt{n_r}$ and $\mathtt{n_1}$ can be found to make the widened post-state become:

$$(4) \vee (\sigma \wedge \mathtt{n_s}{+}\mathtt{n_1}{=}\mathtt{n_r}{-}1 \wedge \mathtt{n_1}{\geq}2) \quad (5)$$

One more iteration of symbolic execution will produce the same result as (5), suggesting that it is already the fixpoint (and hence the loop invariant):

$$\begin{array}{l}
\mathtt{res}{::}\mathtt{sll}\langle \mathtt{n_r}, \mathtt{a}, \mathtt{b}\rangle \wedge \mathtt{srt}{=}\mathtt{res} \wedge \mathtt{prv}{=}\mathtt{null} \wedge \mathtt{n_r}{+}1{=}\mathtt{n}{-}\mathtt{n_x} \vee \\
\mathtt{res}{::}\mathtt{node}\langle \mathtt{a}, \mathtt{srt}\rangle * \mathtt{srt}{::}\mathtt{sll}\langle \mathtt{n_s}, \mathtt{c_1}, \mathtt{b}\rangle \wedge \mathtt{prv}{=}\mathtt{res} \wedge \\
\quad \mathtt{a}{\leq}\mathtt{c_1} \wedge \mathtt{a}{\leq}\mathtt{v} \wedge \mathtt{n_r}{+}1{=}\mathtt{n}{-}\mathtt{n_x} \wedge \mathtt{n_s}{+}1{=}\mathtt{n_r} \vee \\
\mathtt{res}{::}\mathtt{node}\langle \mathtt{a}, \mathtt{prv}\rangle * \mathtt{prv}{::}\mathtt{node}\langle \mathtt{c_1}, \mathtt{srt}\rangle * \mathtt{srt}{::}\mathtt{sll}\langle \mathtt{n_s}, \mathtt{c_2}, \mathtt{b}\rangle \wedge \\
\quad \mathtt{a}{\leq}\mathtt{c_1}{\leq}\mathtt{c_2} \wedge \mathtt{c_1}{\leq}\mathtt{v} \wedge \mathtt{n_r}{+}1{=}\mathtt{n}{-}\mathtt{n_x} \wedge \mathtt{n_s}{+}2{=}\mathtt{n_r} \vee \\
\mathtt{res}{::}\mathtt{sls}\langle \mathtt{prv}, \mathtt{n_1}, \mathtt{a}, \mathtt{c_1}\rangle * \mathtt{prv}{::}\mathtt{node}\langle \mathtt{c_2}, \mathtt{srt}\rangle * \mathtt{srt}{::}\mathtt{sll}\langle \mathtt{n_s}, \mathtt{c_3}, \mathtt{b}\rangle \wedge \\
\quad \mathtt{a}{\leq}\mathtt{c_1}{\leq}\mathtt{c_2}{\leq}\mathtt{c_3} \wedge \mathtt{c_2}{\leq}\mathtt{v} \wedge \mathtt{n_r}{+}1{=}\mathtt{n}{-}\mathtt{n_x} \wedge \mathtt{n_s}{+}\mathtt{n_1}{=}\mathtt{n_r}{-}1 \wedge \mathtt{n_1}{\geq}2
\end{array}$$

Note that although it is possible to further join the third disjunctive branch with the fourth, our analysis does not do so as it tries to keep the result as precise as possible by eliminating only auxiliary pointer variables.

With the frame $\mathtt{cur}{::}\mathtt{node}\langle \mathtt{v}, \mathtt{x}\rangle * \mathtt{x}{::}\mathtt{ll}\langle \mathtt{n_x}\rangle$ added back, the analysis for the outer loop continues until the following (outer) loop invariant is discovered:

$$\begin{array}{l}
(\mathtt{x}{::}\mathtt{ll}\langle \mathtt{n_x}\rangle \wedge \mathtt{res}{=}\mathtt{null} \wedge \mathtt{n_x}{=}\mathtt{n}) \vee (\mathtt{res}{::}\mathtt{node}\langle \mathtt{a}, \mathtt{null}\rangle * \mathtt{x}{::}\mathtt{ll}\langle \mathtt{n_x}\rangle \wedge \mathtt{n}{=}\mathtt{n_x}{+}1) \vee \\
\quad (\mathtt{res}{::}\mathtt{sll}\langle \mathtt{n_r}, \mathtt{a}, \mathtt{b}\rangle * \mathtt{x}{::}\mathtt{ll}\langle \mathtt{n_x}\rangle \wedge \mathtt{n}{=}\mathtt{n_x}{+}\mathtt{n_r} \wedge \mathtt{n_r}{\geq}2)
\end{array}$$

which allows us to verify the whole method successfully using e.g. HIP/SLEEK.

3 Language and Abstract Domain

We focus on a strongly-typed C-like imperative language (in Figure 2, analogous to that in Nguyen et al. [11]) allowing users to define their own data structures and shape predicates. It requires users to provide specifications for methods, but while loops can be annotation-free and we will calculate their invariants.

Prog	::= *tdecl** *meth**	*tdecl*	::= *datat* | *spred*
datat	::= `data` c { *field** }	*field*	::= $t\ v$
meth	::= $t\ mn\ ((t\ v)^*; (t\ v)^*)$ *mspec* $\{e\}$	t	::= $c \mid \tau$
e	::= $d \mid d[v] \mid v{:=}e \mid e_1; e_2 \mid t\ v;\ e \mid$ `if` v `then` e_1 `else` $e_2 \mid$ `while` v $\{e\}$	τ	::= `int` | `bool` | `void`
d	::= `null` $\mid k^{\tau} \mid v \mid$ `new` $c(v^*) \mid mn(u^*; v^*)$		
$d[v]$	::= $v.f \mid v.f{:=}w \mid$ `free`(v)		

Fig. 2. A Core (C-like) Imperative Language.

Our specification language (in Figure 3) allows (user-defined) shape predicates *spred* to specify both shape and numerical properties and lemmas *lemma* for predicate coercion. We require that the predicates be well-founded [11].

$$
\begin{array}{lll}
\mathit{spred} & ::= \texttt{root}{::}c\langle v^*\rangle \equiv \Phi & \Phi ::= \bigvee \sigma^* \qquad \sigma ::= \exists v^* \cdot \kappa \wedge \pi \\
\mathit{mspec} & ::= \mathit{requires}\ \Phi_{pr}\ \mathit{ensures}\ \Phi_{po} & \\
\mathit{lemma} & ::= \texttt{root}{::}c\langle v^*\rangle \wedge \pi \longleftarrow \Phi & \\
\Delta & ::= \Phi \mid \Delta_1 \vee \Delta_2 \mid \Delta \wedge \pi \mid \Delta_1 * \Delta_2 \mid \exists v \cdot \Delta & \\
\kappa & ::= \texttt{emp} \mid v{::}c\langle v^*\rangle \mid \kappa_1 * \kappa_2 & \pi ::= \gamma \wedge \phi \\
\gamma & ::= v_1{=}v_2 \mid v{=}\texttt{null} \mid v_1{\neq}v_2 \mid v{\neq}\texttt{null} \mid \gamma_1 \wedge \gamma_2 & \\
\phi & ::= b \mid a \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \neg\phi \mid \exists v \cdot \phi \mid \forall v \cdot \phi & \\
b & ::= \texttt{true} \mid \texttt{false} \mid v \mid b_1 = b_2 & a ::= s_1{=}s_2 \mid s_1{\leq}s_2 \\
s & ::= k^{\texttt{int}} \mid v \mid k^{\texttt{int}} \times s \mid s_1 + s_2 \mid -s \mid \mathit{max}(s_1,s_2) \mid \mathit{min}(s_1,s_2) &
\end{array}
$$

Fig. 3. The Specification Language.

A conjunctive abstract program state, σ, is composed of a heap (shape) part κ and a numerical part π, where π consists of γ and ϕ as aliasing and numerical information, respectively. We use SH to denote the set of such conjunctive states. During the symbolic execution, the abstract program state at each program point will be a disjunction of σ's, denoted by Δ (and its set is referenced as $\mathcal{P}_{\mathsf{SH}}$). We check the entailment relationship $\vdash$ between two states with Sleek [11].

The memory model of our specification formula is similar to the model given for separation logic [13], except that we have extensions to handle user-defined shape predicates and related numerical properties. In our analysis, all the variables except the program ones are logical variables. We denote a program variable's initial value as unprimed and its current value as primed [11].

4 Abstraction, Join and Widening

Our proposed analysis algorithm is given in Figure 4.

Fixpoint Computation in Combined Domain
Input: $\mathcal{T}$, Δ_{pre}, `while` b $\{e\}$, n;
Local: $i := 0$; $\Delta_i := \texttt{false}$; $\Delta'_i := \texttt{false}$;

```
1  repeat
2    i := i + 1;
3    Δ_i := widen†(Δ_{i−1}, join†(Δ_pre, Δ'_{i−1}));
4    Δ'_i := abs†([[e]]_T(Δ_i ∧ b));
5    if Δ'_i = false ∨ cp_no(Δ'_i) > n
       then return fail end if
6  until Δ'_i = Δ'_{i−1};
7  return Δ'_i
```

Fig. 4. Main analysis algorithm.

As demonstrated in Section 2, our analysis is based on the abstract interpretation framework [4] over a combined shape and numerical domain, by iteration of symbolic executions over the loop body, followed by abstraction over shapes and join and widening over the combined domain. This section concentrates on our specifically designed abstraction, join and widening operations employed in our loop invariant synthesis process.

Abstraction function. During the symbolic execution, we may be confronted with many "concrete" shapes in postconditions of the loop body. As an example of list traversal, the list may contain one node, or two nodes, or even more nodes

in the list, which the analysis cannot enumerate infinitely. The abstraction function deals with those situations by abstracting the (potentially infinite) concrete situations into more abstract shapes, to ensure finiteness over the shape domain. Our rationale is to keep only program variables and a bounded number of shared cutpoints (existentially quantified logical variables referenced by more than one predicates); all other logical variables will be abstracted away. As an instance, the first state below can be further abstracted, while the second one cannot:

$$\begin{array}{l} \mathtt{x}{::}\mathtt{node}\langle \_, \mathtt{z}_0\rangle * \mathtt{z}_0{::}\mathtt{node}\langle \_, \mathtt{null}\rangle \rightsquigarrow \mathtt{x}{::}\mathtt{ll}\langle \mathtt{n}\rangle \wedge \mathtt{n}{=}2 \\ \mathtt{x}{::}\mathtt{node}\langle \_, \mathtt{z}_0\rangle * \mathtt{y}{::}\mathtt{node}\langle \_, \mathtt{z}_0\rangle * \mathtt{z}_0{::}\mathtt{node}\langle \_, \mathtt{null}\rangle \not\rightsquigarrow \end{array} \tag{6}$$

where both $\mathtt{x}$ and $\mathtt{y}$ are program variables, and $\mathtt{z}_0$ is an existentially quantified logical variable. In the second case $\mathtt{z}_0$ is a shared cutpoint referenced by both $\mathtt{x}$ and $\mathtt{y}$, and is therefore preserved. As illustrated, the abstraction transition function abs eliminates unimportant cutpoints (during analysis) to ensure termination. Its type is defined as follows:

$$\mathsf{abs} \ : \ \mathsf{SH} \rightarrow \mathsf{SH} \qquad \text{Abstraction}$$

which indicates that it takes in a conjunctive abstract state σ and abstracts it as another conjunctive state σ' ($\mathsf{abs}(\sigma) \rightsquigarrow \sigma'$). Below are its rules.

$$\mathsf{abs}(\sigma \wedge x_0{=}e) \rightsquigarrow \sigma[e/x_0] \qquad \mathsf{abs}(\sigma \wedge e{=}x_0) \rightsquigarrow \sigma[e/x_0]$$

$$\frac{\neg\mathsf{Reach}(\sigma, x_0)}{\mathsf{abs}(x_0{::}c\langle v^*\rangle * \sigma) \rightsquigarrow \sigma * \mathtt{true}}$$

$$\frac{\begin{array}{c} \mathit{isdatat}(c_1) \qquad c_2\langle u_2^*\rangle \equiv \Phi \\ p{::}c_1\langle v_1^*\rangle * \sigma_1 \vdash p{::}c_2\langle v_2^*\rangle * \sigma_2 \qquad \neg\mathsf{Reach}(p{::}c_2\langle v_2^*\rangle * \sigma_2, v_1^*) \end{array}}{\mathsf{abs}(p{::}c_1\langle v_1^*\rangle * \sigma_1) \rightsquigarrow p{::}c_2\langle v_2^*\rangle * \sigma_2}$$

$$\frac{\begin{array}{c} c_1\langle u_1^*\rangle \equiv \Phi_1 \qquad c_2\langle u_2^*\rangle \equiv \Phi_2 \qquad \mathtt{root}{::}c_2\langle v_2^*\rangle \longleftarrow \mathtt{root}{::}c_1\langle v_1^*\rangle * \sigma \\ p{::}c_1\langle v_1^*\rangle * \sigma_1 \vdash p{::}c_2\langle v_2^*\rangle * \sigma_2 \qquad \neg\mathsf{Reach}(p{::}c_2\langle v_2^*\rangle * \sigma_2, v_1^*) \end{array}}{\mathsf{abs}(p{::}c_1\langle v_1^*\rangle * \sigma_1) \rightsquigarrow p{::}c_2\langle v_2^*\rangle * \sigma_2}$$

The first two rules eliminate logical variables (x_0) by replacing them with their equivalent expressions (e). The third rule is used to eliminate any garbage (heap part led by a logical variable x_0 unreachable from the other part of the heap) that may exist in the heap. As x_0 is already unreachable from, and not usable by, the program variables, it is sound to treat it as garbage $\mathtt{true}$ without losing useful information, for example the $\mathtt{x}_0$ in $\mathtt{x}{::}\mathtt{node}\langle \_, \mathtt{null}\rangle * \mathtt{x}_0{::}\mathtt{node}\langle \_, \mathtt{null}\rangle$ where only $\mathtt{x}$ is a program variable.

The last two rules of abs play the most significant role which intend to eliminate shape formulae led by logical variables. All variables in v_1^* are logical variables. The fourth rule tries to fold a data node up to a predicate node. It confirms that c_1 is a data node definition and c_2 is a predicate. Meanwhile it also ensures that the latter is a sound abstraction of the former by entailment checking, and the logical parameters of c_1 are not reachable from other part of the heap (so that the abstraction does not lose necessary information). Here $\neg\mathsf{Reach}(\sigma, x^*)$ says that no variable in x^* is reachable from any free variable in the abstract state σ. Therefore we only abstract heap parts not led by a shared cutpoint. As the previous example shows, we have $\mathsf{abs}(\mathtt{x}{::}\mathtt{node}\langle \_, \mathtt{z}_0\rangle * \mathtt{z}_0{::}\mathtt{node}\langle \_, \mathtt{null}\rangle) \rightsquigarrow$

$\mathtt{x}{::}\mathtt{ll}\langle \mathtt{n}_0\rangle \wedge \mathtt{n}_0{=}2$. One more note about this is, if we have multiple predicates to choose from, we will have a disjunction of all predicates which are sound as abstraction, such as

$$\begin{array}{l}\mathsf{abs}(\mathtt{x}{::}\mathtt{node}\langle \mathtt{v}_x, \mathtt{z}_0\rangle * \mathtt{z}_0{::}\mathtt{node}\langle \mathtt{v}_z, \mathtt{null}\rangle \wedge \mathtt{v}_x{\leq}\mathtt{v}_z) \;\rightsquigarrow \\ \qquad\qquad \mathtt{x}{::}\mathtt{ll}\langle \mathtt{n}_0\rangle \wedge \mathtt{n}_0{=}2 \vee \mathtt{x}{::}\mathtt{sll}\langle \mathtt{n}_0, \mathtt{v}_x, \mathtt{v}_z\rangle \wedge \mathtt{n}_0{=}2\end{array}$$

The last rule utilises lemmas provided by users to combine two or more predicates. Its basic idea is similar as the previous rule; the extra obligation requires that both c_1 and c_2 be predicates, and a lemma stating their relationship is essential. Sometimes such lemma can bring more power to the abstraction, as it provides more ways to observe a predicate other than its definition. An illustrative case is $\mathsf{abs}(\mathtt{x}{::}\mathtt{ls}\langle \mathtt{z}_0, \mathtt{n}_0\rangle * \mathtt{z}_0{::}\mathtt{ll}\langle \mathtt{n}_1\rangle) \rightsquigarrow \mathtt{x}{::}\mathtt{ll}\langle \mathtt{n}_2\rangle \wedge \mathtt{n}_2{=}\mathtt{n}_0{+}\mathtt{n}_1$, if we are given the lemma to concatenate a list segment with a list to obtain a new list whose length is the sum of the former two.

One more note about abs is that we only allow it to keep a bounded number of shared cutpoints during the analysis. Namely, we will avoid unbounded increment of cutpoints to ensure termination of our analysis. On the basis of this, we apply these rules from the top to the bottom on the abstract state over and again until it stabilises. Such convergence is confirmed because the abstract shape domain is finite due to the combination of bounded variables and predicates, as discussed later.

Join operator. The operator join is applied over two conjunctive abstract states, trying to find a common shape as a sound abstraction for both:

$$\begin{array}{l}\mathsf{join}(\sigma_1, \sigma_2) =_{df} \\ \quad \textbf{let } \sigma_1', \sigma_2' = \mathsf{rename}(\sigma_1, \sigma_2) \textbf{ in} \\ \quad\quad \textbf{match } \sigma_1', \sigma_2' \textbf{ with } (\exists x_1^* \cdot \kappa_1 \wedge \pi_1), (\exists x_2^* \cdot \kappa_2 \wedge \pi_2) \textbf{ in} \\ \quad\quad\quad \textbf{if } \sigma_1 \vdash \sigma_2 * \mathtt{true} \textbf{ then } \exists x_1^*, x_2^* \cdot \kappa_2 \wedge (\mathsf{join}_\pi(\pi_1, \pi_2)) \\ \quad\quad\quad \textbf{else if } \sigma_2 \vdash \sigma_1 * \mathtt{true} \textbf{ then } \exists x_1^*, x_2^* \cdot \kappa_1 \wedge (\mathsf{join}_\pi(\pi_1, \pi_2)) \\ \quad\quad\quad \textbf{else } \sigma_1 \vee \sigma_2\end{array}$$

where the rename function prevents naming clashes among logical variables of σ_1 and σ_2, by renaming logical variables of same name in the two states with fresh names. For example it will renew $\mathtt{x}_0$'s name in both states $\exists \mathtt{x}_0 \cdot \mathtt{x}_0{=}0$ and $\exists \mathtt{x}_0 \cdot \mathtt{x}_0{=}1$ to make them $\exists \mathtt{x}_0 \cdot \mathtt{x}_0{=}0$ and $\exists \mathtt{x}_1 \cdot \mathtt{x}_1{=}1$. After this procedure it judges whether σ_2 is an abstraction of σ_1, or the other way round. If either case holds, it regards the shape of the weaker state as the shape of the joined states, and performs joining for numerical formulae with $\mathsf{join}_\pi(\pi_1, \pi_2)$, the convex hull operator over numerical domain [12]. Otherwise it keeps a disjunction of the two states (as it would be unsound to join their shapes together in this case).

Widening operator. The finiteness of shape domain is confirmed by the abstraction function. To ensure the termination of the whole analysis, we still need to guarantee the convergence over the numerical domain. This task is accomplished by the widening operator.

The widening operator $\mathsf{widen}(\sigma_1, \sigma_2)$ is defined as:

$$\begin{array}{l}\mathsf{widen}(\sigma_1,\sigma_2) =_{df}\\ \quad \textbf{let } \sigma_1',\sigma_2' = \mathsf{rename}(\sigma_1,\sigma_2) \textbf{ in}\\ \quad\quad \textbf{match } \sigma_1',\ \sigma_2' \textbf{ with } (\exists x_1^* \cdot \kappa_1 \wedge \pi_1),\ (\exists x_2^* \cdot \kappa_2 \wedge \pi_2) \textbf{ in}\\ \quad\quad\quad \textbf{if } \sigma_1 \vdash \sigma_2 * \mathtt{true} \textbf{ then } \exists x_1^*, x_2^* \cdot \kappa_2 \wedge (\mathsf{widen}_\pi(\pi_1,\pi_2))\\ \quad\quad\quad \textbf{else } \sigma_1 \vee \sigma_2\end{array}$$

where the rename function has the same effect as above. Generally this operator is analogous to join; the only difference is that the second operand σ_2 should be weaker than the first σ_1 (ensured in previous steps of abstraction and joining), such that the widening reflects the trend of such weakening from σ_1 to σ_2. Then it applies widening operation $\mathsf{widen}_\pi(\pi_1,\pi_2)$ over the numerical domain [12].

These three operations provides termination guarantee while preserving soundness for our analysis. Before using them we also lift them pointwisely from SH to $\mathcal{P}_{\mathsf{SH}}$ such that they can be applied to Δ's as well.

Soundness and termination. The soundness of our analysis is ensured by the soundness of the following: the entailment prover [11], the abstract semantics (w.r.t. concrete semantics), the abstraction operation over shapes, and the join and widening operators.

Lemma 1 (Soundness). *Our analysis is sound due to the soundness of entailment checking, abstract semantics, operations of abstraction, join and widening.*

The proof for entailment checking is by structural induction over abstract domain [11]; for abstract semantics is by induction over program constructors; for abstraction follows directly the first two; and for join and widening is based on entailment checking and soundness of corresponding numerical operators.

For the termination aspect, we have the result:

Lemma 2 (Termination). *The iteration of our fixpoint computation will terminate in finite steps for finite input of program and specification.*

The proof is based on two main facts: the finiteness over shape domain provided by our restriction on cutpoints, and the termination over numerical domain guaranteed by our widening operator. The first can be proved by claiming the finiteness of program and logical variables preserved in our analysis as well as the finiteness of all possible shape predicates, and hence the finiteness of all possible abstract states only with shape information. The second is proved in the abstract interpretation frameworks for numerical domains [12]: as the states over shape domain are finite, the widening operator will guarantee termination for any of these states, and hence the termination of the whole analysis (note that only the abstract states with the same shape will be widened). Meanwhile some other care is also taken for the termination issue, including the termination of entailment checking [11] and lemma application [10].

5 Experiments and Evaluation

We have implemented a prototype system in Objective Caml for evaluation purpose. We used Sleek [11] as the solver for entailment checking over heap domain and Fixcalc solver [12] for join and widening operations in numerical domain. It can be seen from Figure 5 that all programs listed here can be analysed with loop invariants obtained, which are confirmed by Hip/Sleek's verification.

Program	Function	Time
`create`	Creates a list with given length parameter	0.452
`ins_sort`	Inner loop of Fig. 1	0.824
`ins_sort`	Outer loop of Fig. 1	4.372
`delete`	Disposes a list	0.720
`traverse`	Traverses a list	0.636
`append`	Appends two lists	0.312
`partition`	Auxiliary operation used by Quick-sort	1.497
`merge`	Merges two sorted lists to be one sorted list	1.972
`split`	Divides a list into two sublists with length difference of at most one	0.354
`select`	Selects the smallest node of a list	0.692
`select_sort`	Outer loop of selection sort	4.892
`tree_insert`	Inserts a node into a binary search tree	1.364
`tree_search`	Finds a node in a binary search tree	1.294

Fig. 5. Selected Experimental Results.

6 Related Work and Conclusion

Related works. For heap-manipulating programs with any form of recursion (be it loop or recursive method call), dramatic advances have been made in synthesising their invariants/specifications. The local shape analysis [5] infers loop invariants for list-processing programs, followed by the SpaceInvader tool to infer full method specifications over the separation domain, so as to verify pointer safety for larger industrial codes [2, 15]. The SLAyer tool [6] implements an inter-procedural analysis for programs with shape information. To deal with also size information (such as number of nodes in lists/trees), THOR [9] derives a numerical program from the original heap-processing one in a sound way, such that the size information can be obtained with a traditional loop invariant synthesis. A similar approach [7] combines a set domain (for shape) with its cardinality domain (for corresponding numerical information) in a more general framework. Compared with these works, our approach can handle data structures with stronger invariants such as sortedness and binary search property, which have not been addressed in the previous works. One more work to be mentioned is the relational inductive shape analysis [3]. It employs inductive checkers to express both shape and numerical information. Our advantage over theirs is that we try to keep as many as possible shared cutpoints (logical variables) during the analysis (within a preset bound), whereas they do not preserve such cutpoints (which is witnessed by their joining rules over the shape domain). Therefore our analysis is essentially more precise than theirs, e.g. in the second scenario of (6) described in Section 4. Meanwhile their "checkers" do not cover data structures with loops (such as cyclic lists) while we can handle them naturally. Lastly, their work is mainly from a theory perspective as they do not employ numerical reasoners to solve the relational constraints in their implementation; comparatively, we discharge all the numerical and relational constraints with automated reasoners [12].

There are also other approaches that can synthesise shape-related program invariants than those based on separation logic, e.g., the shape analysis frame-

work TVLA [14] based on three-valued logic. It is capable of handling complicated data structures and properties, such as sortedness. Compared with their work, our separation-logic-based approach also unifies heterogeneous techniques and annotations in a homogeneous way with predicates which capture sufficient information to perform analysis of properties that involve closures. Thanks to separation logic, our approach benefits from the frame rule and hence supports local reasoning (only the footprint of a program is considered when the analysis if performed), which fact enhances its scalability.

Concluding Remarks. We have reported an analysis which allows us to synthesise sound and useful loop invariants over a combined separation and numerical domain. The key components of our analysis include novel operations for abstraction, join and widening in the combined domain. Our next step is to conduct more experimental results to further confirm its viability. Meanwhile we will try to eliminate the request of users to provide a limit of cutpoint bounds by more thorough investigations into this arbitrary user-defined specification mechanism.

Acknowledgement. This work was supported in part by EPSRC projects [EP/E021948/1, EP/G042322/1]. We thank Florin Craciun's precious comments.

References

1. J. Berdine, C. Calcagno, and P. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, 2005.
2. C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In *36th POPL*, January 2009.
3. B. Chang and X. Rival. Relational inductive shape analysis. In *POPL*, 2008.
4. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.
5. D. Distefano, P. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS*, 2006.
6. A. Gotsman, J. Berdine, and B. Cook. Interprocedural shape analysis with separated heap abstractions. In *SAS*, 2006.
7. S. Gulwani, T. Lev-Ami, and M. Sagiv. A Combination Framework for Tracking Partition Sizes. In *POPL*, 2009.
8. S. Ishtiaq and P. O'Hearn. BI as an assertion language for mutable data structures. In *POPL*, 2001.
9. S. Magill, M. Tsai, P. Lee, and Y. Tsay. Thor: A tool for reasoning about shape and arithmetic. In *CAV*, 2008.
10. H. H. Nguyen and W.-N. Chin. Enhancing program verification with lemmas. In *20th CAV*, 2008.
11. H. H. Nguyen, C. David, S. Qin, and W.-N. Chin. Automated verification of shape and size properties via separation logic. In *8th VMCAI*, 2007.
12. C. Popeea and W.-N. Chin. Inferring disjunctive postconditions. In *Proceedings of 11th Asian Computing Science Conference*, 2006.
13. J. Reynolds. Separation logic: a logic for shared mutable data structures. In *17th LICS*, 2002.
14. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, 2002.
15. H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. O'Hearn. Scalable shape analysis for systems code. In *20th CAV*, 2008.

Relational program logics and self-composition

Lennart Beringer

Department of Computer Science, Princeton University,
35 Olden Street, Princeton NJ 08540
eberinge@cs.princeton.edu

Abstract. Relational program logics are a powerful tool for the verification of program transformations and security properties, and enable extensional interpretations of type systems and program analyses. We present a rule-by-rule encoding of a termination-insensitive variant of Benton's Relational Hoare Logic (RHL) in a unary program logic, demonstrating that standard verification technology may actually suffice for the formal certification of such relational properties. Our embedding employs a reformulation of self-composition at the level of program logics and has been formally verified by an implementation in Isabelle/HOL.

1 Introduction

Relational program logics have emerged as a tool for verifying program transformations, interpreting security analyses, and relating data structure implementations [7, 1, 22]. They expose the two-execution nature of program properties at the level of judgements, extending traditional Hoare/VDM-style unary program logics where a judgement's interpretation refers to a single execution. Benton [7] makes a particularly compelling argument for relational reasoning, motivating it both from a theoretical perspective and from a compiler writer's point of view. In particular, Benton demonstrates that relational Hoare logic (RHL) may serve as a declarative interface between program analyses and the transformations enabled by these analyses.

Given the advent of relational logics, it is reasonable to ask whether relational logics are actually more expressive than unary logics. In this paper, we give a negative answer to this question, for a variant of Benton's RHL. We show that relational proof rules can be derived from a pair of unary specifications whose derivation trees are independent.

In addition to being of theoretical interest, our result opens an avenue for integrating relational logics into architectures for certified compilation and proof-carrying code. As has been argued in [4, 2], a modular way to build such systems is to develop stacks of program logics which mediate between machine-level logics and program analyses. Each layer being derived from its host by a formal interpretation, the soundness of the entire architecture rests on that of its bottom-most layer. Here, sophisticated semantic models are currently being developed [3] which justify the foundational layers with respect to operational semantics of mainstream programming languages such as C [16]. Given the conceptual and engineering complexity inherent in these models, it is desirable to leverage their construction (in particular: their implementation in theorem provers) over a wide range of application logics. Our results indicate that it may indeed be possible to avoid to repeat the construction for relational properties.

With this scenario in mind, we exploratively formalized our work in the theorem prover Isabelle/HOL [8]. For our simple programming language, a standard soundness result for the host logic suffices, allowing us to postpone the integration with a more sophisticated model. As a side effect, this implementation provides to our knowledge the first formalized soundness proof of a relational program logic. Like most formalizations of program logics developed since Kleymann's thesis [15] we use a shallow embedding for the assertion language but a deep one for the programming language.

Our variant of RHL differs from Benton's original as follows. First, we interpret judgements in a *termination-insensitive* way, deeming two programs vacuously equivalent if either one fails to terminate. This discipline is more coarse-grained than denotational equivalence as considered by Benton but separates functional equivalence from the largely orthogonal issue of termination. Second, our encoding treats the program phrases in a slightly nonsymmetric way. As a consequence of these design decisions, Benton's specialised rules for pre- and postrelations that are partial equivalence relations are either unsound (rule of transitivity) or require the exploitation of the formal completeness of the host logic (rule of symmetry). As our unary host logic is largely syntax-directed, its completeness is also required for deriving relational proof rules that reverse the standard syntactic compositionality discipline, i.e. rules whose concluding judgement concerns a subphrase of the phrases occurring in the hypotheses. An advantage of the termination-insensitive setting is that an injection rule for pairs of unary judgements can be derived. Finally, we give some novel rules, in particular a rule for comparing loops that do not proceed in lock-step.

Outline of technical contribution Our encoding generalizes the approach presented in [10], where we show how to lift Barthe et al.'s technique of *self-composition* [6] from the syntactic level to the level of program logics. Self-composition solves the certification problem for noninterference of some program C by reducing it to a one-execution property of the program $C; C'$. Here, C' arises from C by replacing all program variables x in C by fresh copies x'. The reformulation of self-composition in [10] defines a family of noninterference-guaranteeing program logic assertions which replace the syntactic code duplication by a corresponding operation at the level of judgements. These assertions are parameterized over correlation formulae ϕ, which communicate information between the two copies of the self-composed program, intuitively serving as an abstraction of the state at the program point ; where C and C' are composed. Moreover, suitable correlations ϕ for nonatomic phrases can be synthesized from the correlations ϕ_i associated with their subphrases, in a rule-by-rule fashion following type systems for noninterference [21, 13]. Thus, noninterference-guaranteeing assertions may be obtained automatically from these type systems. Summarizing, logical self-composition is the type-directed calculation of a unary program specification for noninterference, and avoids the code duplication and variable renaming required by syntactic self-composition. We summarize the work from [10] in Section 2.

The encoding of the present paper refines the class of noninterference-guaranteeing assertions by exploiting the fact that each such assertion is in fact obtained as a combination of two subspecifications. Thinking in terms of syntactic self-composition, the first (left) subspecification concerns the behaviour of the original program C while the second (right) subspecification abstracts the behaviour of the renamed copy C'. By

exposing the separation into left and right subspecifications, we enable their application to different programs, as is required for modeling the two-program judgements of RHL. Indeed, this separation also applies to general "relation-transforming" rather than "noninterference-guaranteeing" assertions. Significantly, the relation-transforming assertions may again be parameterized by correlation witnesses ϕ. The synthesis of these witnesses proceeds in many cases exactly as in the case of noninterference, factoring into derivations for left and right subspecifications. Thus, each RHL judgement is the conjunction of two unary judgements that may be independently derived and are only coordinated by the use of the shared correlation witness ϕ.

Like Kleymann in his formal treatment of auxiliary variables [15], we employ universal quantification over states to link initial conditions to states in later points of execution. However, our quantification concerns states in the opposite program/execution and occurs inside (relational) assertions whereas in Kleymann's case the quantification concerns a single execution and occurs in the semantic interpretation of judgements.

The Isabelle-formalization underlying this paper is available for download at [8].

2 Self-composition for simple noninterference

In this section we summarize the treatment of noninterference from [10].

Programming language We consider the well-known language **IMP** of assignments and loops, defined over sets $\mathcal{X}$ of variables (ranged over by x) and $\mathcal{V}$ of values (ranged over by v). We employ a big-step relation $\sigma \xrightarrow{C} \tau$ where $\sigma, \tau, \ldots$ are from the set $\Sigma \equiv \mathcal{X} \to \mathcal{V}$ of states. We write $[\![.]\!]_\sigma$ to denote the evaluation of an arithmetic or boolean expression in state σ. The (unsurprising) rules defining the semantics are omitted.

Unary axiomatic semantics We define an axiomatic semantics for partial correctness. As programs cannot get stuck, we employ judgements in precondition-less VDM style, where assertions A are binary state relations, i.e. are of type $\mathcal{A} \equiv \Sigma \to \Sigma \to \mathcal{B}$. Here, $\mathcal{B}$ denotes the set of (meta-level) booleans. Judgement take the form $\rhd C : A$, i.e. relate a program phrase C with a specification. Command C *satisfies* an assertion, notation $\models C : A$, if for all σ and τ, $\sigma \xrightarrow{C} \tau$ implies $A\ \sigma\ \tau$.

The proof rules are defined in the table below – the final two rules were not considered in [10] but are useful for deriving the rules in Section 3.

Hypotheses & side conditions	*Conclusion*
	$\rhd \mathbf{Skip} : \lambda\ \sigma\ \tau.\ \tau = \sigma$
	$\rhd x{:=}e : \lambda\ \sigma\ \tau.\ \tau = \sigma[x \mapsto [\![e]\!]_\sigma]$
$\rhd C : A; \quad \rhd D : B$	$\rhd C;D : \lambda\ \sigma\ \tau.\ \exists\ \rho.\ A\ \sigma\ \rho \wedge B\ \rho\ \tau$
$\rhd C : A; \quad \rhd C' : A'$ $B = \lambda\ \sigma\ \tau.\ ([\![b]\!]_\sigma \Rightarrow A\ \sigma\ \tau) \wedge (\neg[\![b]\!]_\sigma \Rightarrow A'\ \sigma\ \tau)$	$\rhd \mathbf{If}\ b\ \mathbf{then}\ C\ \mathbf{else}\ C' : B$
$\rhd C : B; \quad \forall\ \sigma.\ \neg[\![b]\!]_\sigma \Rightarrow A\ \sigma\ \sigma$ $\forall\ \sigma\ \rho\ \tau.\ [\![b]\!]_\sigma \Rightarrow B\ \sigma\ \rho \Rightarrow A\ \rho\ \tau \Rightarrow A\ \sigma\ \tau$	$\rhd \mathbf{While}\ b\ \mathbf{do}\ C : \lambda\ \sigma\ \tau.\ A\ \sigma\ \tau \wedge \neg[\![b]\!]_\tau$
$\rhd C : A; \quad \forall\ \sigma\ \tau.\ A\ \sigma\ \tau \Rightarrow B\ \sigma\ \tau$	$\rhd C : B$
	$\rhd C : \lambda\ \sigma\ \tau.\ True$
$\rhd C : A \qquad \rhd C : B$	$\rhd C : \lambda\ \sigma\ \tau.\ A\ \sigma\ \tau \wedge B\ \sigma\ \tau$

Using standard techniques [15, 18], it is easy to show soundness and completeness of the proof system (relative to the ambient logic HOL), i.e. the following property.

Theorem 1. *(Soundness and completeness of program logic) The derivability of the judgement $\rhd C : A$ is equivalent to $\models C : A$, i.e., its semantic validity.*

Noninterference and self-composition Consider the following notions of state indistinguishability and termination-insensitive noninterference ("security"), where the set $\mathcal{X}$ of program variables is statically partitioned into high security variables $\mathcal{X}_{high}$ and low security variables $\mathcal{X}_{low}$.

Definition 1. *Two states $\sigma, \tau \in \Sigma$ are* indistinguishable *(w.r.t. low variables), written $\sigma \sim \tau$, if $\sigma(x) = \tau(x)$ for all $x \in \mathcal{X}_{low}$. Program C is* secure *if whenever $\sigma \sim \sigma'$ and $\sigma \xrightarrow{C} \tau$ and $\sigma' \xrightarrow{C} \tau'$ then $\tau \sim \tau'$.*

Syntactic self-composition [6] verifies that program C is secure by showing that $\models C_1;C_2 : A_{\mathsf{ssc}}$ holds (for example by showing that $\rhd C_1;C_2 : A_{\mathsf{ssc}}$ is derivable), where C_1 and C_2 are copies of C operating on mutually disjoint sets of variables, $A_{\mathsf{ssc}} = \lambda(\sigma_1, \sigma_2)\ (\tau_1, \tau_2).\sigma_1 \sim \sigma_2 \Rightarrow \tau_1 \sim \tau_2$, and we write $\sigma = (\sigma_1, \sigma_2)$ whenever σ_i is the part of σ that refers to the variables in C_i and similarly for τ.

The *(program-)logical* account of self-composition mimics the executions of C_1 and C_2 in a single VDM assertion for C. This assertions consists of two conjuncts, the first corresponding to the execution of C_1, the second corresponding to the execution of C_2. The conjuncts are linked by a relation ϕ of type $\mathcal{T} \equiv (\Sigma \times \Sigma) \to \mathcal{B}$. This witness ϕ plays the role of an assertion applicable at the point of self-composition, when C_1 has finished execution but C_2 has not yet started, i.e. at state (τ_1, σ_2) of an execution

$$(\sigma_1, \sigma_2) \xrightarrow{C_1} (\tau_1, \sigma_2) \xrightarrow{C_2} (\tau_1, \tau_2).$$

Definition 2 (Sec). *For $\phi : \mathcal{T}$ we let Sec(ϕ) denote the assertion*

$$\mathit{Sec}(\phi) \equiv \lambda\, \sigma\, \tau.\, (\forall\, \rho.\ \sigma \sim \rho \Rightarrow \phi(\tau, \rho))\ \wedge\ (\forall\, \rho.\ \phi(\rho, \sigma) \Rightarrow \rho \sim \tau)$$

The first conjunct mimics the execution $(\sigma_1, \sigma_2) \xrightarrow{C_1} (\tau_1, \sigma_2)$: applying $\mathit{Sec}(\phi)$ to $\sigma_1 \xrightarrow{C} \tau_1$ and instantiating ρ with σ_2 yields $\phi(\tau_1, \sigma_2)$ whenever $\sigma_1 \sim \sigma_2$. Similarly, the second conjunct mimics the execution $(\tau_1, \sigma_2) \xrightarrow{C_2} (\tau_1, \tau_2)$. Indeed, applying $\mathit{Sec}(\phi)$ to $\sigma_2 \xrightarrow{C} \tau_2$ and instantiating ρ with τ_1 yields $\tau_1 \sim \tau_2$ whenever $\phi(\tau_1, \sigma_2)$. Combining the two parts, we thus have $\tau_1 \sim \tau_2$ whenever $\sigma_1 \sim \sigma_2$, as desired.

This reasoning we just gave amounts to the proof of the following soundness result.

Proposition 1. *If $\models C : \mathit{Sec}(\phi)$ then C is secure.*

For proving C secure, it thus suffices to exhibit an arbitrary ϕ with $\rhd C : \mathit{Sec}(\phi)$. In principle, this strategy *always* succeeds: we also showed in [10] that if C is secure then the witness constructed as $\phi \equiv \lambda\ (\tau, \sigma').\ \exists \sigma.\ \sigma \xrightarrow{C} \tau \wedge \sigma \sim \sigma'$ always satisfies $\models C : \mathit{Sec}(\phi)$. This completeness result suggests in particular that any static analysis

$$\text{T-ExpH}\frac{}{\vdash e : \mathit{high}} \quad \text{T-ExpL}\frac{\mathcal{X}_{\mathit{high}} \cap \mathit{Vars}(e) = \emptyset}{\vdash e : \mathit{low}} \quad \text{T-BExp}\frac{\vdash e_1 : i \quad \vdash e_2 : i}{\vdash e_1 \ \mathit{bop}\ e_2 : i}$$

$$\text{T-Skip}\ \frac{}{[\mathit{high}] \vdash \mathbf{Skip}}\ \text{T-AssH}\frac{}{[\mathit{high}] \vdash h{:=}e}\ \text{T-AssL}\frac{\vdash e : \mathit{low}}{[\mathit{low}] \vdash l{:=}e} \quad \text{T-Sub}\ \frac{[\mathit{high}] \vdash C}{[\mathit{low}] \vdash C}$$

$$\text{T-Comp}\frac{[i] \vdash C_1 \quad [i] \vdash C_2}{[i] \vdash C_1;C_2}\ \text{T-Whl}\ \frac{\vdash b : i \quad [i] \vdash C}{[i] \vdash \mathbf{While}\ b\ \mathbf{do}\ C} \quad \text{T-If}\ \frac{\vdash b : i \quad [i] \vdash C_1 \quad [i] \vdash C_2}{[i] \vdash \mathbf{If}\ b\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2}$$

Fig. 1. Volpano-Smith-Irvine Type System. Security levels: $i \in \{\mathit{low}, \mathit{high}\}$; Variables: $h \in \mathcal{X}_{\mathit{high}}, l \in \mathcal{X}_{\mathit{low}}$, Judgement forms: $\vdash e : i$, $\vdash b : i$, and $[i] \vdash C$

that ensures noninterference of C yields the existence of a witness. This insight can be confirmed in a constructive manner, taking, for example, the type system of Volpano, Smith, and Irvine [21], as follows. Figure 1 summarizes the type system in the presentation of Sabelfeld and Myers [19], restricted to the lattice $\mathit{low} \sqsubset \mathit{high}$. The rules guarantee security as follows:

Proposition 2. *If* $\vdash e : \mathit{low}$ *then* $\sigma \sim \tau \Rightarrow [\![e]\!]_\sigma = [\![e]\!]_\tau$. *If* $\vdash b : \mathit{low}$ *then* $\sigma \sim \tau \Rightarrow [\![b]\!]_\sigma = [\![b]\!]_\tau$. *If* $[\mathit{low}] \vdash C$ *then* C *is secure. If* $[\mathit{high}] \vdash C$ *then* $\sigma \xrightarrow{C} \tau$ *implies* $\sigma \sim \tau$.

In order to obtain the desired relations ϕ along with derivations of $\rhd\ C : \mathit{Sec}(\phi)$ automatically from typing derivations, we show that each typing axiom yields a witness ϕ for its concluding judgement, and that the rules for composite statement forms combine the witnesses associated with the hypothetical judgements to witnesses for the concluding judgements. We interpret a typing judgement $[\mathit{low}] \vdash C$ as a judgement $\rhd\ C : \mathit{Sec}(\phi)$ for some suitable ϕ, and a typing judgement $[\mathit{high}] \vdash C$ as a judgement $\rhd\ C : \mathit{HiSec}$, where *HiSec* abbreviates the assertion $\lambda\ \sigma\ \tau.\ \sigma \sim \tau$. The translation of typing rules into VDM derivations of security assertions is shown in Figure 2. In particular, the rules constructively exhibit assertions ϕ for the commands typeable in a low context. All rules are derivable from the VDM logic.

Theorem 2. *Let* $[i] \vdash C$. *If* $i = \mathit{high}$ *then* $\rhd\ C : \mathit{HiSec}$ *holds, and if* $i = \mathit{low}$ *then there is some* ϕ *such that* $\rhd\ C : \mathit{Sec}(\phi)$.

3 RHL in decomposed form

We generalize the development from Section 2 towards RHL by separating the two conjuncts of $\mathit{Sec}(\phi)$ in Definition 2. Indeed,

$$\mathit{Sec}(\phi) = \lambda\ \sigma\ \tau.\ \mathit{Sec}_{\mathsf{L}}(\phi)\ \sigma\ \tau \wedge \mathit{Sec}_{\mathsf{R}}(\phi)\ \sigma\ \tau,$$

where $\mathit{Sec}_{\mathsf{L}}(\phi) \equiv \lambda\ \sigma\ \tau.\ \forall\ \rho.\ \sigma \sim \rho \Rightarrow \phi(\tau, \rho)$ and $\mathit{Sec}_{\mathsf{R}}(\phi) \equiv \lambda\ \sigma\ \tau.\ \forall\ \rho.\ \phi(\rho, \sigma) \Rightarrow \rho \sim \tau$. We can thus understand each judgement $\rhd\ C : \mathit{Sec}(\phi)$ in Figure 2 as the conjunction of judgements $\rhd\ C : \mathit{Sec}_{\mathsf{L}}(\phi)$ and $\rhd\ C : \mathit{Sec}_{\mathsf{R}}(\phi)$.

$$\textsc{AssignH}\ \frac{}{\rhd\ h{:=}e : \mathit{HiSec}} \qquad \textsc{AssignL}\ \frac{\forall\,\sigma\,\tau.\ \sigma \sim \tau \Rightarrow [\![e]\!]_\sigma = [\![e]\!]_\tau}{\rhd\ l{:=}e : \mathit{Sec}(\lambda\ (\sigma,\tau).\ \sigma \sim \tau[l \mapsto [\![e]\!]_\tau])}$$

$$\textsc{CompL}\ \frac{\rhd\ C_1 : \mathit{Sec}(\phi) \qquad \rhd\ C_2 : \mathit{Sec}(\psi)}{\rhd\ C_1;C_2 : \mathit{Sec}(\lambda\ (\sigma,\tau).\ \exists\,\rho.\ \phi(\rho,\tau) \wedge (\forall\,\omega.\ \rho \sim \omega \Rightarrow \psi(\sigma,\omega)))}$$

$$\textsc{CompH}\ \frac{\rhd\ C_1 : \mathit{HiSec} \qquad \rhd\ C_2 : \mathit{HiSec}}{\rhd\ C_1;C_2 : \mathit{HiSec}} \qquad \textsc{IfH}\ \frac{\rhd\ C_1 : \mathit{HiSec} \quad \rhd\ C_2 : \mathit{HiSec}}{\rhd\ \mathbf{If}\ b\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2 : \mathit{HiSec}}$$

$$\textsc{IfL}\ \frac{\forall\,\sigma\,\tau.\ \sigma \sim \tau \Rightarrow [\![b]\!]_s = [\![b]\!]_\tau \quad \rhd\ C_1 : \mathit{Sec}(\phi) \quad \rhd\ C_2 : \mathit{Sec}(\psi)}{\rhd\ \mathbf{If}\ b\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2 : \mathit{Sec}(\lambda\ (\sigma,\tau).\ ([\![b]\!]_\tau \Rightarrow \phi(\sigma,\tau)) \wedge (\neg[\![b]\!]_\tau \Rightarrow \psi(\sigma,\tau)))}$$

$$\textsc{WhileH}\ \frac{\rhd\ C : \mathit{HiSec}}{\rhd\ \mathbf{While}\ b\ \mathbf{do}\ C : \mathit{HiSec}} \qquad \textsc{Sub}\ \frac{\rhd\ C : \mathit{HiSec}}{\rhd\ C : \mathit{Sec}(\lambda\ (\sigma,\tau).\ \sigma \sim \tau)}$$

$$\textsc{SkipH}\ \frac{}{\rhd\ \mathbf{Skip} : \mathit{HiSec}} \qquad \textsc{WhileL}\ \frac{\forall\,\sigma\,\sigma'.\ \sigma \sim \sigma' \Rightarrow [\![b]\!]_\sigma = [\![b]\!]_{\sigma'} \qquad \rhd\ C : \mathit{Sec}(\phi)}{\rhd\ \mathbf{While}\ b\ \mathbf{do}\ C : \mathit{Sec}(\widehat{\mathcal{F}_{(b,\phi)}})}$$

$\widehat{\mathcal{F}_{(b,\phi)}}$ is the least fixed point of the operator

$$\psi \longmapsto \lambda(\sigma,\tau).(\neg[\![b]\!]_\tau \Rightarrow \sigma \sim \tau) \wedge ([\![b]\!]_\tau \Rightarrow (\exists\,\rho.\phi(\rho,\tau) \wedge (\forall\,\omega.\ \rho \sim \omega \Rightarrow \psi(\sigma,\omega)))).$$

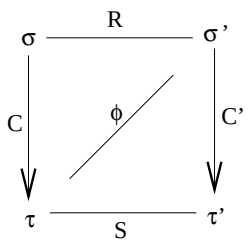
Fig. 2. Derived proof rules for base-line noninterference.

Significantly, this factoring is compatible with the structure of the proof rules and with the synthesis of the witnesses ϕ. To see this, define the proof system $\rhd_{\mathsf{L}}\ C : A$ for the left conjuncts by replacing $\mathit{Sec}(\phi)$ by $\mathit{Sec}_{\mathsf{L}}(\phi)$ throughout the rules of Figure 2, and define the proof system $\rhd_{\mathsf{R}}\ C : A$ for the right conjuncts similarly, by replacing $\mathit{Sec}(\phi)$ by $\mathit{Sec}_{\mathsf{R}}(\phi)$. We then have the following:

Proposition 3. *The systems $\rhd_{\mathsf{L}}\ C : A$ and $\rhd_{\mathsf{R}}\ C : A$ are separately derivable in the logic of Section 2.*

As the next step, we apply the judgements in Proposition 3 to different phrases C and C', whose derivations share only the witness ϕ. Observing that the above reasoning is independent from indistinguishability, we finally arrive at assertions

$$\mathit{RSec}_{\mathsf{L}}(R,\phi) \equiv \lambda\ \sigma\ \tau.\ \forall\rho'.\ \sigma R \rho' \Rightarrow \phi(\tau,\rho')$$
$$\mathit{RSec}_{\mathsf{R}}(S,\phi) \equiv \lambda\ \sigma'\ \tau'.\ \forall\rho.\ \phi(\rho,\sigma') \Rightarrow \rho S \tau'.$$



In particular, note that states in the first position of ϕ refer to the execution on the left and that states in the second position refer to the execution on the right – a discipline that is less apparent from the nonseparated assertion[1] $\mathit{Sec}(.)$.

[1] This observation suggests that by employing different unary logics on both sides one can relate programs that stem from different languages or that use different notions of state. Following

The corresponding generalization of security is

Definition 3. *Let R, S be relational predicates on states and C and C' programs. We write $R : C \approx C' : S$ if $\tau S \tau'$ holds whenever $\sigma R \sigma'$ and $\sigma \xrightarrow{C} \tau$ and $\sigma' \xrightarrow{C'} \tau'$.*

This interpretation coincides with Benton's interpretation of RHL judgements $C \sim C' : R \Rightarrow S$, except that the latter additionally requires equitermination and is formulated in denotational rather than operational terms.

The following is the analogon to Proposition 1 and the completeness result from [10].

Proposition 4. *Let ϕ be such that $\models C : \mathit{RSec}_{\mathsf{L}}(R, \phi)$ and $\models C' : \mathit{RSec}_{\mathsf{R}}(S, \phi)$. Then $R : C \approx C' : S$ holds. Conversely, if $R : C \approx C' : S$ then $\phi_{C,R} = \lambda(\rho, \rho').\ \exists \sigma.\ \sigma \xrightarrow{C} \rho \wedge \sigma R \rho'$ satisfies $\models C : \mathit{RSec}_{\mathsf{L}}(R, \phi_{C,R})$ and $\models C' : \mathit{RSec}_{\mathsf{R}}(S, \phi_{C,R})$.*

We encode Benton's judgements $C \sim C' : R \Rightarrow S$ as suggested by Proposition 4, i.e. as pairs of judgements $\rhd C : \mathit{RSec}_{\mathsf{L}}(R, \phi)$ and $\rhd C' : \mathit{RSec}_{\mathsf{R}}(S, \phi)$ for suitable ϕ. In order to obtain a compact presentation, we abbreviate these judgements as $\phi \vdash_{\mathsf{L}} C \prec C' : R \Rightarrow S$ and $\phi \vdash_{\mathsf{R}} C \prec C' : R \Rightarrow S$, respectively [2]. The rules of RHL may be categorized as follows.

Core homogeneous rules relate phrases of identical syntactic structure – their embeddings are shown in Figure 3. The witnesses ϕ are precisely those from Figure 2, modulo the replacement of $\sim$ by general relations R, S, ... In rule WHL , $\widehat{\mathcal{F}_{(b',R,\phi)}}$ is defined as the least fixed point of the (monotone in ψ) operator

$$\mathcal{F}_{(b',R,\phi)} : \psi \longmapsto \lambda(\sigma, \tau).\ (\neg[\![b']\!]_\tau \Rightarrow \sigma R \tau) \wedge ([\![b']\!]_\tau \Rightarrow (\exists\, \rho.\phi(\rho, \tau) \wedge (\forall \omega.\rho R \omega \Rightarrow \psi(\sigma, \omega)))).$$

Benton's original rules may be obtained from our rules by erasing the witnesses ϕ and replacing $\prec$ by $\sim$.

Logical rules relate arbitrary phrases. We have derived Benton's rules of subtyping (which amounts to the rule of consequence) and falsity, and novel but hardly surprising rules for intersecting and unioning witnesses. The latter are useful for deriving some of the other rules given, and yield trivial RHL rules if we erase the witnesses ϕ. We omit the rules due to space limitations.

Transformation rules relate phrases of syntactically different shape. Figure 4 shows the left-rules for eliminating dead code (modeled as equivalence to $\mathbf{Skip}$) and common branches. The former specializes to the rules for dead assignments and dead loops from [7]. Starting with this rule, we freely mix unary and relational hypotheses: unary hypotheses $\rhd C : A$ essentially amount to relational hypotheses $\phi \vdash_\circ C \prec \mathbf{Skip} : R \Rightarrow S$ or $\phi \vdash_\circ \mathbf{Skip} \prec C' : R \Rightarrow S$ but in contrast to the latter can be exploited in cases where $\mathbf{Skip}$ does not occur syntactically in the conclusion[3]. We omit the rules for similar operations on the right program phrase.

---

the submission of the present article, we have indeed developed decomposition for arbitrary pairs of (possibly nondeterministic) transition systems. See reference [9].

[2] This combination into a single parameterized proof system results in a minor loss of precision: in some of the rules, some side conditions are only used for one of the two systems.

[3] The effect observed here is exactly the one from Kleymann's formalization of auxiliary state. Indeed, unary Hoare logics with auxiliary state can be immediately modeled in a relational style precisely by using the diagonal precondition and the silent instruction $\mathbf{Skip}$.

$$\text{SKIP}\ \frac{\phi = \lambda\,(\sigma,\tau).\ \sigma R \tau}{\phi \vdash_\circ \mathbf{Skip} \prec \mathbf{Skip} : R \Rightarrow R}$$

$$\text{ASS}\ \frac{R = \{(\sigma,\sigma') \mid (\sigma[x \mapsto [\![e]\!]_\sigma])S(\sigma'[x' \mapsto [\![e']\!]_{\sigma'}])\} \quad \phi = \lambda\,(\sigma,\tau).\ \sigma S(\tau[x' \mapsto [\![e']\!]_\tau])}{\phi \vdash_\circ x{:=}e \prec x'{:=}e' : R \Rightarrow S}$$

$$\text{SEQ}\ \frac{\begin{array}{c}\phi = \lambda\,(\sigma,\tau).\ \exists\,\rho.\ \phi_1(\rho,\tau) \wedge (\forall\,\omega.\ \rho T \omega \Rightarrow \phi_2(\sigma,\omega)) \\ \phi_1 \vdash_\circ C \prec C' : R \Rightarrow T \qquad \phi_2 \vdash_\circ D \prec D' : T \Rightarrow S\end{array}}{\phi \vdash_\circ C;D \prec C';D' : R \Rightarrow S}$$

$$\text{IF}\ \frac{\begin{array}{c}R = \{(\sigma,\sigma') \mid \sigma T \sigma' \wedge [\![b]\!]_\sigma = [\![b']\!]_{\sigma'}\} \qquad R_1 = \{(\sigma,\sigma') \mid \sigma T \sigma' \wedge [\![b]\!]_\sigma \wedge [\![b']\!]_{\sigma'}\} \\ R_2 = \{(\sigma,\sigma') \mid \sigma T \sigma' \wedge \neg[\![b]\!]_\sigma \wedge \neg[\![b']\!]_{\sigma'}\} \\ \phi_1 \vdash_\circ C \prec C' : R_1 \Rightarrow S \qquad \phi_2 \vdash_\circ D \prec D' : R_2 \Rightarrow S \\ \phi = \lambda\,(\sigma,\tau).\ ([\![b']\!]_\tau \Rightarrow \phi_1(\sigma,\tau)) \wedge (\neg[\![b']\!]_\tau \Rightarrow \phi_2(\sigma,\tau))\end{array}}{\phi \vdash_\circ \mathbf{If}\ b\ \mathbf{then}\ C\ \mathbf{else}\ D \prec \mathbf{If}\ b'\ \mathbf{then}\ C'\ \mathbf{else}\ D' : R \Rightarrow S}$$

$$\text{WHL}\ \frac{\begin{array}{c}R = \{(\sigma,\sigma') \mid \sigma T \sigma' \wedge [\![b]\!]_\sigma = [\![b']\!]_{\sigma'}\} \qquad S = \{(\sigma,\sigma') \mid \sigma T \sigma' \wedge \neg([\![b]\!]_\sigma \vee [\![b']\!]_{\sigma'})\} \\ U = \{(\sigma,\sigma') \mid \sigma T \sigma' \wedge [\![b]\!]_\sigma \wedge [\![b']\!]_{\sigma'}\} \qquad \phi \vdash_\circ C \prec C' : U \Rightarrow R\end{array}}{\widehat{\mathcal{F}_{(b',R,\phi)}} \vdash_\circ \mathbf{While}\ b\ \mathbf{do}\ C \prec \mathbf{While}\ b'\ \mathbf{do}\ C' : R \Rightarrow S}$$

Fig. 3. Embedded syntactic core rules of RHL. All rules carry the implicit side condition $\circ \in \{\mathsf{L}, \mathsf{R}\}$. An alternative choice for ϕ in rule ASS is $\phi = \lambda(\sigma,\tau).\ \exists\,\rho.\ \sigma = \rho[x \mapsto [\![e]\!]_\rho]) \wedge \rho R \tau$.

$$\text{CBR-L}\ \frac{\begin{array}{c}\phi_1 \vdash_\circ C_1 \prec C' : R_1 \Rightarrow S \qquad \phi_2 \vdash_\circ C_2 \prec C' : R_2 \Rightarrow S \\ R_1 = \{(\sigma,\sigma').\ \sigma R \sigma' \wedge [\![b]\!]_\sigma\} \qquad R_2 = \{(\sigma,\sigma').\ \sigma R \sigma' \wedge \neg[\![b]\!]_\sigma\} \\ \phi = \lambda\,(\sigma,\tau).\ \phi_1(\sigma,\tau) \vee \phi_2(\sigma,\tau)\end{array}}{\phi \vdash_\circ \mathbf{If}\ b\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2 \prec C' : R \Rightarrow S}$$

$$\text{DEADL}\ \frac{\rhd C : A \qquad \forall\,\sigma\,\sigma'\,\tau.\ A\,\sigma\,\tau \Rightarrow \sigma R \sigma' \Rightarrow \tau S \sigma' \qquad \phi = \lambda\,(\sigma,\tau).\ \sigma S \tau}{\phi \vdash_\circ C \prec \mathbf{Skip} : R \Rightarrow S}$$

Fig. 4. Selected transformation rules. Both rules carry the implicit side condition $\circ \in \{\mathsf{L}, \mathsf{R}\}$.

Figure 5 shows the remaining rules from [7]. In the absence of side conditions that stipulate that relations be partial equivalence relations, transitivity requires

$$\{(\sigma,\sigma'').\ \exists\,\sigma'.\ \sigma R\sigma' \wedge \sigma' U\sigma''\} : C \approx C'' : \{(\sigma,\sigma'').\ \exists\,\sigma'.\ \sigma T\sigma' \wedge \sigma' S\sigma''\}$$

to hold whenever $R : C \approx C' : T$ and $U : C' \approx C'' : S$. As related phrases are not necessarily equiterminating, this implication is only satisfied if we impose an appropriate side condition, for example by requiring that R and the termination of C guarantee the termination of C':

$$\forall\,\sigma\,\sigma'.\ \sigma R\sigma' \Rightarrow (\exists\,\tau.\ \sigma \xrightarrow{C} \tau) \Rightarrow (\exists\,\tau'.\ \sigma' \xrightarrow{C'} \tau').$$

The variant of symmetry appropriate in the absence of PER-assumptions concerns the equivalence of $R : C \approx C' : S$ and $R^{-1} : C' \approx C : S^{-1}$ which indeed holds, as Proposition 4 implies that $\models C : \mathit{RSec}_{\mathsf{L}}(R,\phi) \wedge \models C' : \mathit{RSec}_{\mathsf{R}}(S,\phi)$ is equivalent to $\models C' : \mathit{RSec}_{\mathsf{L}}(R^{-1},\phi_{C',R^{-1}}) \wedge \models C : \mathit{RSec}_{\mathsf{R}}(S^{-1},\phi_{C',R^{-1}})$. Due to the formal asymmetry in the decomposition, however, $R : C \approx C' : S$ and $R^{-1} : C' \approx C : S^{-1}$ can only be turned into judgements in the unary logic if we exploit the formal completeness result. The nonderivability of R-CBINVTL and R-CBINVFL in their decomposed form is unsurprising, too: these rules reverse the syntactic subphrase relationship between hypotheses and conclusions that is largely followed by our unary program logic. Again, both rules are semantically sound, hence we can formally derive them using completeness.

$$\text{R-SYM}\ \frac{C \sim C' : R \Rightarrow S \qquad \mathit{Per}(R \Rightarrow S)}{C' \sim C : R \Rightarrow S} \qquad \text{R-TR}\ \frac{C \sim C' : R \Rightarrow S \qquad C' \sim C'' : R \Rightarrow S \qquad \mathit{Per}(R \Rightarrow S)}{C \sim C'' : R \Rightarrow S}$$

$$\text{R-CBINVTL}\ \frac{\mathbf{If}\ b\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2 \sim C : R \Rightarrow S \qquad T = \{(\sigma,\sigma').\ \sigma R\sigma' \wedge [\![b]\!]_\sigma\}}{C_1 \sim C : T \Rightarrow S}$$

$$\text{R-CBINVFL}\ \frac{\mathbf{If}\ b\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2 \sim C : R \Rightarrow S \qquad T = \{(\sigma,\sigma').\ \sigma R\sigma' \wedge \neg[\![b]\!]_\sigma\}}{C_2 \sim C : T \Rightarrow S}$$

Fig. 5. Remaining RHL rules

Finally, we have derived some rules that are not given in [7] – see Figure 6.

Rule UNARY injects two unary judgements. Its specialization to a single Hoare-style hypothesis $\{P\}\ C\ \{Q\}$ (i.e. to the case where $C = C'$, $R = P \times P$ and $S = Q \times Q$ with $A = A' = \lambda\,\sigma\,\tau.\ P\,\sigma \to Q\,\tau$) is discussed in [7] but argued to be unsound as C may in general exhibit different termination behaviour in two states satisfying P. Our termination-insensitive setting admits the generalized rule.

Rule IF-DIFF applies if the branch conditions b and b' evaluate differently. It may in fact be possible to derive IF-DIFF from the earlier rules but this has not been proven yet, at least in the absence of PER-ness assumptions. Combining IF-DIFF with IF yields the expected rule for arbitrary conditionals (not shown).

Finally, rule WHLNOLKSTP relates two loops, but in contrast to rule WHL from Figure 3, the iterations do not have to proceed in lock-step. Instead, each loop body has to preserve the relational invariant, for arbitrary fixed states of the opposite execution. The asymmetry in the side conditions for A and A' is a consequence of the fact that our embedding prioritizes the left phrase. Clearly, the rule that omits the negatively occurring $\neg[\![b]\!]_\sigma$ from the side condition on A' is derivable from the rule given.

$$\text{UNARY}\ \frac{\begin{array}{c}\rhd C : A \qquad \phi = \lambda\,(\sigma,\tau).\,\exists\,\rho.\,A\;\rho\;\sigma \wedge \rho R \tau \qquad \rhd\ C' : A' \\ R = \{(\sigma,\sigma').\,\forall\;\tau\;\tau'.\,A\;\sigma\;\tau \Rightarrow A'\;\sigma'\;\tau' \Rightarrow \sigma S \sigma'\}\end{array}}{\phi \vdash_\circ C \prec C' : R \Rightarrow S}$$

$$\text{IF-DIFF}\ \frac{\begin{array}{c}\phi_1 \vdash_\circ C \prec D' : U \Rightarrow S \qquad \phi_2 \vdash_\circ D \prec C' : V \Rightarrow S \\ U = \{(\sigma,\sigma').\,\sigma T \sigma' \wedge [\![b]\!]_\sigma \wedge \neg[\![b']\!]_{\sigma'}\} \qquad V = \{(\sigma,\sigma').\,\sigma T \sigma' \wedge \neg[\![b]\!]_\sigma \wedge [\![b']\!]_{\sigma'}\} \\ R = \{(\sigma,\sigma').\,\sigma T \sigma' \wedge \neg[\![b]\!]_\sigma = [\![b']\!]_{\sigma'}\} \\ \phi = \lambda\,(\sigma,\tau).\,([\![b']\!]_\tau \Rightarrow \phi_2(\sigma,\tau)) \wedge (\neg[\![b']\!]_\tau \Rightarrow \phi_1(\sigma,\tau))\end{array}}{\phi \vdash_\circ \mathbf{If}\ b\ \mathbf{then}\ C\ \mathbf{else}\ D \prec \mathbf{If}\ b'\ \mathbf{then}\ C'\ \mathbf{else}\ D' : R \Rightarrow S}$$

$$\text{WHLNOLKSTP}\ \frac{\begin{array}{c}\phi \vdash_\circ C \prec C' : T \Rightarrow R \qquad \rhd\ C : A \qquad \rhd\ C' : A' \\ T = \{(\sigma,\sigma').\,\sigma R \sigma' \wedge [\![b]\!]_\sigma \wedge [\![b']\!]_{\sigma'}\} \\ \forall\;\sigma\;\tau\;\sigma'.\,A\;\sigma\;\tau \Rightarrow \sigma R \sigma' \Rightarrow [\![b]\!]_\sigma \Rightarrow \tau R \sigma' \\ \forall\;\sigma'\;\tau'\;\sigma.\,A'\;\sigma'\;\tau' \Rightarrow \sigma R \sigma' \Rightarrow [\![b']\!]_{\sigma'} \Rightarrow \neg[\![b]\!]_\sigma \Rightarrow \sigma R \tau' \\ S = \{(\sigma,\sigma').\,\sigma R \sigma' \wedge \neg[\![b]\!]_\sigma \wedge \neg[\![b']\!]_{\sigma'}\} \\ \psi = \lambda\,(\sigma,\tau).(\sigma R \tau \wedge \neg[\![b]\!]_\sigma) \vee \\ \qquad \exists\,\rho.\,\left(\begin{array}{l}\neg[\![b']\!]_\tau \wedge \neg[\![b]\!]_\sigma \wedge (\forall \rho'.\,\rho R \rho' \Rightarrow \sigma R \rho') \wedge \\ (\forall\,\omega.\,\neg[\![b]\!]_\omega \Rightarrow (\forall\,\rho'.\,\rho R \rho' \Rightarrow \omega R \rho') \Rightarrow (\omega R \tau \wedge \neg[\![b']\!]_\tau))\end{array}\right)\end{array}}{\psi \vdash_\circ \mathbf{While}\ b\ \mathbf{do}\ C \prec \mathbf{While}\ b'\ \mathbf{do}\ C' : R \Rightarrow S}$$

Fig. 6. Additional rules. All rules carry the implicit side condition $\circ \in \{\mathsf{L}, \mathsf{R}\}$.

Summarizing our embedding we have the following:

Theorem 3. *The rules in Figures 3, 4, and 6 are derivable in the VDM logic: whenever* $\phi \vdash_\circ C \prec C' : R \Rightarrow S$ *is derivable using these rules, then* $\circ = \mathsf{L}$ *implies* $\rhd C :$ *RSec*$_\mathsf{L}(R, \phi)$*, and* $\circ = \mathsf{R}$ *implies* $\rhd C' :$ *RSec*$_\mathsf{R}(S, \phi)$*.*

Combining this property with Proposition 4 and Theorem 1 yields that $R : C \approx C' : S$ holds whenever there is some ϕ such that both $\phi \vdash_\mathsf{L} C \prec C' : R \Rightarrow S$ and $\phi \vdash_\mathsf{R} C \prec C' : R \Rightarrow S$ are derivable.

4 Discussion

Barthe et al.'s syntactic self-composition extended earlier work by Joshi and Leino [14]. Terauchi and Aiken [20] push the self-composition operation towards the leaves of the syntax tree using program transformations, guided by noninterference type systems.

The resulting verification conditions are reported to be amenable to fully automated methods in many cases. Our synthesis of witnesses amounts to a syntax- rather than type-based decomposition. We have not yet explored the automated discharge of side conditions and expect the potential for this to be limited, due to the expressiveness of the encoded logic. Independently from [6], Darvas et al. [11] developed a variation of self-composition in dynamic logic, with similar goals in mind as [10].

A natural question is whether our technique could be generalized to other language features and alternative interpretations of judgements. In work performed since the submission of the present paper, we have extended the embedding to an object-based language, including an operational semantics with explicit error states. We have subsequently derived relational and unary separation logics, in a termination-insensitive but fault-avoiding interpretation [9]. We plan to integrate simple procedures in the near future, along the lines of [10]. The consideration of termination-sensitivity is future work.

Barthe et al. also present a variation of self-composition where a parallel composition operator is used in place of the sequential operator. The self-composed program is then verified using CTL. Transferring these ideas to a setting of relational logic appears difficult – indeed, we are unaware of any relational temporal logic in the literature.

Separation and concurrency are both supported by the Cminor stack of program logics currently being developed at Princeton. As mentioned in the introduction, a long-term goal is to integrate relational logics into this system and subsequently to justify high-level analyses and transformations. As a first step towards this integration we formalized an embedding of the VDM logic in a Hoare-style logic with separate pre- and postconditions.

The main purpose of the present work being the formal embedding of the existing RHL we have not explored concrete verifications that make use of the novel proof rules. Such an evaluation might include the verification of some loop transformations routinely employed in optimizing compilers [5]. Transformation rules that relate multiple (e.g. nested) loops have already been developed in the translation validation community [12]. At present, it is unclear whether these proofs may be reformulated using our technique. On the other hand, the completeness of our interpretation (i.e. the second half of Proposition 4) ensures that our system can in principle be extended by any proof rule that comes with an operational soundness proof.

Funding information This work is funded in part by the Air Force Office of Scientific Research (FA9550-09-1-0138) and by the National Science Foundation (CNS-0910448).

References

1. T. Amtoft, S. Bandhakavi, and A. Banerjee. A logic for information flow in object-oriented programs. In Morrisett and Jones [17], pages 91–102.
2. A. W. Appel. Foundational high-level static analysis. In *Proceedings of the CAV 2008 Workshop on Exploiting Concurrency Efficiently and Correctly (EC^2)*, July 2008.
3. A. W. Appel and S. Blazy. Separation logic for small-step Cminor. In *Theorem Proving in Higher Order Logics: Proceedings of the 20th International Conference TPHOLs 2007*, Lecture Notes in Computer Science 4732, pages 5–21. Springer-Verlag, 2007.

4. D. Aspinall, L. Beringer, M. Hofmann, H.-W. Loidl, and A. Momigliano. A program logic for resources. *Theoretical Computer Science*, 389(3):411–445, 2007.
5. D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, 1994.
6. G. Barthe, P. D'Argenio, and T. Rezk. Secure Information Flow by Self-Composition. In R. Foccardi, editor, *Computer Security Foundations Workshop*, pages 100–114. IEEE Press, 2004.
7. N. Benton. Simple relational correctness proofs for static analyses and program transformations. In N. D. Jones and X. Leroy, editors, *Proceedings of the 31st ACM Symposium on Principles of Programming Languages, POPL'04, Venice, Italy*, pages 14–25. ACM Press, 2004.
8. L. Beringer. Relation program logics and self-composition – Isabelle formalization. Available from `www.cs.princeton.edu/~eberinge`, 2010.
9. L. Beringer. Relational program logics in decomposed style. Submitted. Available from `www.cs.princeton.edu/~eberinge`, 2010.
10. L. Beringer and M. Hofmann. Secure information flow and program logics. In *IEEE Computer Security Foundations Symposium*, pages 233–248. IEEE Press, 2007.
11. A. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In R. Gorrieri, editor, *Workshop on Issues in the Theory of Security*. IFIP WG 1.7, ACM SIGPLAN and GI FoMSESS, 2003.
12. Y. Hu, C. Barrett, and B. Goldberg. Theory and algorithms for the generation and validation of speculative loop optimizations. In *Proceedings of the* 2^{nd} *IEEE International Conference on Software Engineering and Formal Methods (SEFM '04)*, pages 281–289. IEEE Computer Society, Sept. 2004. Beijing, China.
13. S. Hunt and D. Sands. On flow-sensitive security types. In Morrisett and Jones [17], pages 79–90.
14. R. Joshi and K. R. M. Leino. A semantic approach to secure information flow. *Science of Computer Programming*, 37(1-3):113–138, 2000.
15. T. Kleymann. *Hoare Logic and VDM: Machine-Checked Soundness and Completeness Proofs*. PhD thesis, LFCS, University of Edinburgh, 1998.
16. X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.
17. J. G. Morrisett and S. L. P. Jones, editors. *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*. ACM Press, 2006.
18. T. Nipkow. Hoare logics for recursive procedures and unbounded nondeterminism. In J. Bradfield, editor, *Computer Science Logic*, volume 2471 of *Lecture Notes in Computer Science*, pages 103–119. Springer-Verlag, 2002.
19. A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communication*, 21:5–19, Jan. 2003.
20. T. Terauchi and A. Aiken. Secure information flow as a safety problem. In C. Hankin and I. Siveroni, editors, *Static Analysis Symposium*, volume 3672 of *Lecture Notes in Computer Science*, pages 352–367. Springer-Verlag, 2005.
21. D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.
22. H. Yang. Relational separation logic. *Theoretical Computer Science*, 375(1-3):308–334, 2007.

VeriFast: Imperative Programs as Proofs

Bart Jacobs*, Jan Smans, and Frank Piessens

Department of Computer Science, Katholieke Universiteit Leuven, Belgium
{bart.jacobs,jan.smans,frank.piessens}@cs.kuleuven.be

Abstract. This paper describes the VeriFast prototype program verification tool, which implements a separation-logic-based approach for the specification and verification of safety properties of pointer-manipulating imperative programs. The approach's distinctive feature is that it combines very good and predictable verification performance with powerful proofs written conveniently as part of the program. We describe the tool's support for the C language.
The paper introduces the tool's various features by means of a running example of a linked list implementation. A detailed formalization of the core of the approach and a soundness proof are available on the website.

1 Introduction

VeriFast is a research prototype program verification tool for verification of safety properties of C and Java programs, based on separation logic.

The safety properties to be verified are specified as annotations in the source code, in the form of function preconditions and postconditions expressed as separation logic assertions. To enable rich specifications, the user may include additional annotations that define inductive datatypes, primitive recursive pure functions over these datatypes, and abstract predicates (i.e. named, parameterized assertions). Abstract predicates may be recursive. A restricted form of existential quantification is supported in assertions in the form of pattern matching.

Verification is based on forward symbolic execution, where memory is represented as a separate conjunction of points-to assertions and abstract predicate assertions, and data values are represented as first-order logic terms with a set of constraints. Abstract predicates must be folded and unfolded explicitly using ghost statements. Rewritings of the abstract state that require induction, or derivations of facts over data values that require induction, can be done by defining *lemma functions*, which are like ordinary C functions except that it is checked that they terminate. Specifically, when a lemma function performs a recursive call, either the recursive call must apply to a strict subset of memory, or one of its parameters must be an inductive value whose size decreases at each recursive call.

Assertions over data values are delegated to an SMT solver, formulated as queries against an axiomatization of the inductive datatypes and recursive pure

* Bart Jacobs is a Postdoctoral Fellow of the Research Foundation - Flanders (FWO)

functions. Importantly, no exhaustiveness axioms are included in this axiomatization; this prevents the SMT solver from performing case analysis on inductive values. Combined with a measure to prevent infinite reductions due to *self-feeding recursions*, this ensures termination of the SMT solver.

A prototype implementation, a formalization and soundness proof, a tutorial text, and a large number of example annotated programs are available at http://www.cs.kuleuven.be/~bartj/verifast/. The implementation includes an IDE that enables the user to step through a failed execution trace, inspecting the symbolic state at each step.

2 Walkthrough

We introduce the approach through an example annotated C program that implements a linked list ADT. Successive figures show successive fragments of the example program.

2.1 Symbolic execution and predicates

VeriFast performs symbolic execution. The symbolic state consists of three parts: the symbolic heap, the symbolic store, and the path condition. The symbolic heap is a bag of *chunks*. A chunk consists of a predicate name and an argument list. Chunk arguments are terms of first-order logic. The predicate name may be the predicate corresponding to a struct field, in which case we call the chunk a *points-to chunk*. A points-to chunk has two arguments: a term denoting the address of the struct, and a term denoting the field value. The predicate name may also refer to a user-defined predicate.

The symbolic store maps local variable names to terms that denote the variable's current value. The path condition is a set of formulae of first-order logic. These constrain the interpretation of the logical symbols used in the terms in the symbolic store and the symbolic heap.

In the implementation, the terms are SMT solver terms, and the path condition corresponds to the state of the SMT solver. The verifier pushes formulae into the SMT solver during symbolic execution, and pops them when a branch of symbolic execution is finished and the next branch is started.

Notice that this means that the SMT solver does not come into contact with the heap. The heap is dealt with syntactically within the verifier itself. This avoids the quantified formulae required by verification condition generation-based approaches to describe heap effects.

Figure 1 shows function *createNode*. It uses an abstract predicate [16] to hide the internal layout of a node. The **close** ghost statement removes the points-to chunks for the individual fields of n from the symbolic heap, and adds a *node* abstract predicate chunk, as expected by the postcondition. The asterisk denotes separating conjunction: $P * Q$ holds if the heap can be split into two separate parts such that P holds for one part and Q for the other. As we will see, operationally, in our tool the asterisk means sequential composition of consumption

```
struct node { struct node *next; int value; };

predicate node(struct node *n, struct node *next, int value) =
  n→next ↦ next * n→value ↦ value * malloc_block_node(n);

struct node *create_node(struct node *next, int value)
  requires emp; ensures node(result, next, value);
{
  struct node *n := malloc(sizeof(struct node));
  n→next := next; n→value := value;
  close node(n, next, value); return n;
}
```

Fig. 1. Example demonstrating abstract predicates and ghost statements (Note: annotations are shown on a gray background. Also, for readability, we typeset some operators differently from the implementation.)

or production of assertions. That is, consuming $P * Q$ means first consuming P and then Q; producing $P * Q$ means first producing P and then Q.

When symbolically executing function *create_node*, the successive symbolic states are as follows:

```
// h = ∅, s = {next ↦ next, value ↦ value}, Σ = ∅
struct node *n := malloc(sizeof(struct node));
// h = {node_next(n, value0), node_value(n, value1), malloc_block_node(n)},
// s = {n ↦ n, next ↦ next, value ↦ value}, Σ = {n ≠ 0}
n→next := next; n→value := value;
// h = {node_next(n, next), node_value(n, value), malloc_block_node(n)},
// s = {n ↦ n, next ↦ next, value ↦ value}, Σ = {n ≠ 0}
close node(n, next, value);
// h = {node(n, next, value)}
// s = {n ↦ n, next ↦ next, value ↦ value}, Σ = {n ≠ 0}
```

Italic names denote program variables; sans serif names denote logical symbols. At the start of symbolic execution, fresh symbols are generated for the function arguments. In the example, fresh symbols are also generated by the *malloc* call to denote *malloc*'s return value and the initial values of the newly allocated fields.

Figure 2 shows a way to denote a piece of memory containing a set of consecutive nodes. Specifically, abstract predicate $lseg(n1, n2, v)$ represents a set of consecutive nodes where the first node is at *n1* and the last node's next pointer points to *n2*, and the nodes store the list of integers v. As a special case, if *n1* equals *n2*, the predicate denotes the empty piece of memory.

During symbolic execution of a function, assertions are *produced* and *consumed*. Producing a points-to assertion or an abstract predicate assertion means adding the corresponding chunk to the symbolic heap, and consuming it means

```
inductive list = nil | cons(int, list);
predicate lseg(struct node *n1, struct node *n2, list v) =
  n1 = n2 ? v = nil : node(n1, ?n, ?h) * lseg(n, n2, ?t) * v = cons(h, t);
```

Fig. 2. Example demonstrating inductive datatype definitions, recursive abstract predicates, conditional assertions, and pattern matching

removing a matching chunk from the symbolic heap. If no matching chunk is present in the symbolic heap, an error is reported. If the assertion being consumed contains patterns, the matching process binds the pattern variables; their scope includes the rest of the assertion, or if the pattern occurs in a function body or precondition, its scope includes the rest of the function. Producing a pure assertion (i.e., a boolean expression), means adding it to the path condition, and consuming it means asking the SMT solver to check that it follows from the current path condition. Producing or consuming a separate conjunction means first producing, resp. consuming the first operand, and then producing, resp. consuming the second operand. Producing or consuming **emp** does nothing. If during execution of a function, a conditional construct is encountered, then the remainder of the execution is performed once for each branch of the construct, after adding the corresponding constraint to the path condition. The conditional constructs include the if and switch statements, the if-then-else assertions, and the switch assertions.

Execution of a function starts with an empty symbolic heap and an empty path condition. Then, the precondition is produced. Then, each statement is executed. And finally, the postcondition is consumed. If subsequently, any chunks are left in the symbolic heap, this is considered a potential memory leak and an error is reported. Execution of a function call statement proceeds by first consuming the call's precondition and then producing its postcondition. Execution of an open ghost statement proceeds by first consuming the abstract predicate assertion and then producing its body. Execution of a close ghost statement proceeds by first consuming the predicate's body and then adding the abstract predicate chunk. Patterns may be used as abstract predicate arguments in an open statement, but in the current implementation they cannot be used as arguments in a close statement.

Figure 3 shows the first part of the client-visible interface of the linked list ADT. The implementation keeps a sentinel node at the end of the list, and it keeps a pointer to the first node and to this sentinel node. The dummy patterns (_) in the definition of the *llist* predicate indicate that the *next* and *value* fields of the sentinel node are insignificant.

```
struct llist { struct node *first; struct node *last; };

predicate llist(struct llist *l, list v) =
  l→first ↦ ?fn * l→last ↦ ?ln * lseg(fn, ln, v) * node(ln, _, _) * malloc_block_llist(l);

struct llist *create_llist()
  requires emp;
  ensures llist(result, nil);
{
  struct llist *l := malloc(sizeof(struct llist));
  struct node *n := create_node(0, 0); l→first := n; l→last := n;
  close lseg(n, n, nil); close llist(l, nil);
  return l;
}
```

Fig. 3. Example demonstrating dummy patterns

```
lemma void distinct_nodes(struct node *n1, struct node *n2)
  requires node(n1, ?n1n, ?n1v) * node(n2, ?n2n, ?n2v);
  ensures node(n1, n1n, n1v) * node(n2, n2n, n2v) * n1 ≠ n2;
{
  open node(n1, _, _); open node(n2, _, _);
  close node(n1, n1n, n1v); close node(n2, n2n, n2v);
}
```

Fig. 4. Example demonstrating lemma functions, distinctness constraint production and patterns in open statements

2.2 Lemma functions

Figure 4 shows a *lemma function*, which is like a C function except that it is declared in an annotation and the verifier checks that it terminates and that it has no effect on memory (i.e. it does not allocate, free, or write to memory). The only effect of calling a lemma function is that it rewrites the symbolic heap into a semantically equivalent but syntactically different one, and/or that it adds constraints to the path condition.

In this example, there is no net change to the symbolic heap; all the lemma function does is add a constraint. Specifically, given two nodes for which there are separate abstract predicate chunks in the symbolic heap, the lemma produces a constraint that says that the nodes are distinct.

Such distinctness constraints are not produced automatically by the verifier for abstract predicate chunks, since the fact that two abstract predicate chunks referring to the same abstract predicate appear in memory does not imply any-

thing about distinctness of the arguments. However, the verifier produces them for points-to chunks. Specifically, when producing a points-to assertion $t{\rightarrow}f \mapsto v$, then for any existing points-to chunk $t'{\rightarrow}f \mapsto v'$ in the symbolic heap, a constraint $t \neq t'$ is added automatically. In the example, this occurs during execution of the second open statement.

```
fixpoint list add(list v, int x) {
  switch (v) {
    case nil : return cons(x, nil);
    case cons(h, t) :
      return cons(h, add(t, x));
  }
}

lemma add_lemma(struct node *n1,
    struct node *n2, struct node *n3)
  requires lseg(n1, n2, ?v)
    * node(n2, n3, ?x) * node(n3, _, _);
  ensures lseg(n1, n3, add(v, x))
    * node(n3, _, _);
{
  distinct_nodes(n2, n3);
  open lseg(n1, _, _);
  if (n1 = n2) {
    close lseg(n3, n3, nil);
  } else {
    distinct_nodes(n1, n3);
    open node(n1, ?n1n, ?n1v);
    add_lemma(n1n, n2, n3);
    close node(n1, n1n, n1v);
  }
  close lseg(n1, n3, add(v, x));
}
```

```
void add(struct llist *l, int x)
  requires llist(l, ?v);
  ensures llist(l, add(v, x));
{
  open llist(l, v);
  struct node *n := create_node(0, 0);
  struct node *nl := l→last;
  open node(nl, _, _);
  nl→next := n;
  nl→value := x;
  close node(nl, n, x);
  l→last := n;
  struct node *nf := l→first;
  add_lemma(nf, nl, n);
  close llist(l, add(v, x));
}
```

Fig. 5. Example demonstrating fixpoint functions and recursive lemma functions

Figure 5 shows the second client-visible list ADT function, function *add*. It adds a value to the end of the list. Its contract describes its effect on the ADT's abstract value using the *fixpoint function add*. (Note that fixpoint function names and non-fixpoint (i.e., regular or lemma) function names are in separate namespaces; the former may occur only in expressions in annotations, whereas the latter may occur only in call statements.)

A fixpoint function is not allowed to read or modify memory. Its body must be a switch statement over one of the function's parameters. We call this parameter the function's *inductive parameter*. The body of each clause of the switch

statement must be a return statement. A fixpoint function may call other fixpoint functions, but not regular functions or lemma functions. Furthermore, to ensure termination, any call must either be a call of a fixpoint function declared earlier in the program, or it must be a direct recursive call where the argument for the inductive parameter is a variable bound by the switch statement.

Regular function *add* creates a new node to serve as the new sentinel node, then updates the old sentinel node's fields, and finally calls the lemma function *add_lemma* to merge the old sentinel node into the *lseg* abstract predicate chunk. The lemma function does so using recursion. Lemma functions may perform recursive calls, but only direct recursive calls, and termination is ensured by checking that at each recursive call either the size of the piece of memory that the function operates on decreases (specifically, after consuming the precondition there must be a points-to chunk left in the symbolic heap), or, similar to a fixpoint function, the function's body is a switch statement over one of its parameters, and the argument for the inductive parameter in the recursive call is bound by this switch statement. Note that a recursive lemma constitutes an inductive proof of the fact that the precondition implies the postcondition.

```
int removeFirst(struct llist *l)
  requires llist(l, ?v) * v ≠ nil; ensures llist(l, ?t) * v = cons(result, t);
{
  open llist(l, v);
  struct node *nf := l→first; open lseg(nf, ?nl, v); open node(nf, _, _);
  struct node *nfn := nf→next; int nfv := nf→value; free(nf); l→first := nfn;
  open lseg(nfn, nl, ?t); close lseg(nfn, nl, t); close llist(l, t);
  return nfv;
}
```

Fig. 6. Example demonstrating execution splits due to conditional constructs in the bodies of predicates being opened

2.3 Case splits

Figure 6 shows a function that removes the first element from a list. It requires that the list is non-empty. When the *lseg* starting at *nf* is opened, an execution split occurs because the body of this predicate is an if-then-else assertion. The verifier notices immediately that the then branch is infeasible and does not continue execution on this branch.

Notice also that the first node is freed. A statement *free*(p); looks for a chunk of the form *malloc_block_*$T(p)$, and then for points-to chunks of the form $p{\rightarrow}f \mapsto v$, for each field f of T, and it removes all of these chunks.

```
void dispose(struct llist *l)
  requires llist(l, _);
  ensures emp;
{
  open llist(l, _);
  struct node *n := l→first;
  struct node *nl := l→last;
  while (n ≠ nl)
    invariant lseg(n, nl, _);
  {
    open lseg(n, nl, _);
    open node(n, _, _);
    struct node *next := n→next;
    free(n);
    n := next;
  }
  open lseg(n, n, _);
  open node(l, _, _);
  free(nl);
  free(l);
}
```

Fig. 7. Example demonstrating loops

```
void main()
  requires emp;
  ensures emp;
{
  struct llist *l := create_llist();
  add(l, 10);
  add(l, 20);
  add(l, 30);
  add(l, 40);
  int x0 := removeFirst(l);
  assert(x0 = 10);
  int x1 := removeFirst(l);
  assert(x1 = 20);
  dispose(l);
}
```

Fig. 8. Example client program for the list ADT

2.4 Loops

Figure 7 shows function *dispose*, which takes a list of arbitrary length. It first frees all proper nodes, then it removes the remaining empty *lseg* assertion, then it frees the sentinel node, and finally it frees the **struct** *llist* object itself. A loop invariant must be provided for each loop. Execution of a loop proceeds by first consuming the loop invariant, assigning fresh symbols to the locals modified by the loop body, and producing the loop invariant again. Then execution proceeds along two branches: in one branch, the loop condition is produced, then the loop body is executed, and finally the loop invariant is consumed. If any chunks remain, this is considered a leak error. In the other branch, first the negation of the loop condition is produced, and then execution proceeds after the loop statement.

Figure 8 wraps up the example by showing an example client program for the list ADT. This program verifies; it follows that all assert statements succeed and no memory is leaked.

Notice that the SMT solver successfully evaluates the *add* fixpoint function applications.

3 Performance

The time complexity of verification is unbounded in theory. Specifically, since recursive pure functions of arbitrary time complexity may be defined, there is no bound on the time complexity of SMT queries. Furthermore, the approach, as currently implemented, does not perform joining of symbolic execution paths after conditional constructs; therefore, the number of symbolic execution steps is exponential in the number of such constructs. However, since no significant search is performed implicitly by the verifier or the SMT solver, performance is very good in practice.

The table below shows indicative verification times for a few example programs.

program	total # lines	# annotation lines	time taken (seconds)
chat server	242	114	0.08
linked list and iterator	332	194	0.09
composite	345	263	0.09
JavaCard applet	340	95	0.51
GameServer	383	148	0.23

4 Related work

Reynolds [17] introduced separation logic. Smallfoot [3] is a tool that performs symbolic execution using separation logic. This technique has been extended for greater automation [19], for termination proofs [4, 6], for fine-grained concurrency [5], for lock-based concurreny [11], and for Java [8, 12]. Unlike VeriFast, all of these tools attempt to infer loop invariants automatically.

Alternative specification and verification approaches, based on generation of verification conditions instead of symbolic execution, include VCC [7], Caduceus [9], ESC-Java [10], KeY [2], Jahob [20], regional logic [1], and approaches based on dynamic frames [13] including VeriCool [18], Dafny [14], and Chalice [15].

5 Conclusion

We presented an approach for specification and verification of imperative programs, that combines very good and predictable verification performance with powerful proofs written conveniently as part of the program. We are currently working to increase the degree of automation while preserving these strengths.

Bibliography

[1] Anindya Banerjee, David A. Naumann, and Stan Rosenberg. Regional logic for local reasoning about global invariants. In *ECOOP*, 2008.

[2] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt. *Verification of Object-Oriented Software: The KeY Approach*. Springer, 2007.
[3] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, 2006.
[4] James Brotheston, Richard Bornat, and Cristiano Calcagno. Cyclic proofs of program termination in separation logic. In *POPL*, 2008.
[5] Cristiano Calcagno, Matthew Parkinson, and Viktor Vafeiadis. Modular safety checking for fine-grained concurrency. In *SAS*, 2007.
[6] Byron Cook, Alexey Gotsman, Andreas Podelski, Andrey Rybalchenko, and Moshe Y. Vardi. Proving that programs eventually do something good. In *POPL*, 2007.
[7] Markus Dahlweid, Michał Moskal, Thomas Santen, Stephan Tobies, and Wolfram Schulte. VCC: Contract-based modular verification of concurrent C. In *ICSE*, 2009.
[8] Dino Distefano and Matthew Parkinson. jStar: Towards practical verification for Java. In *OOPSLA*, 2008.
[9] Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *CAV*, 2007.
[10] Cormac Flanagan, K. Rustan, M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *PLDI*, 2002.
[11] Alexey Gotsman, Josh Berdine, Byron Cook, Noam Rinetzky, and Mooly Sagiv. Local reasoning for storable locks and threads. In *APLAS*, 2007.
[12] Christian Haack and Clément Hurlin. Separation logic contracts for a Java-like language with fork/join. In *AMAST*, 2008.
[13] Ioannis T. Kassios. Dynamic frames: support for framing, dependencies and sharing without restrictions. In *FM*, 2006.
[14] K. Rustan M. Leino. Specification and verification of object-oriented software: Marktoberdorf international summer school 2008 pre-lecture notes, 2008.
[15] K. Rustan M. Leino, Peter Müller, and Jan Smans. *Foundations of Security Analysis and Design V, FOSAD 2007/2008/2009 Tutorial Lectures*, volume 5705 of *LNCS*, chapter Verification of concurrent programs with Chalice. Springer, 2009.
[16] Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. In *POPL*, 2005.
[17] J. C. Reynolds. Separation logic: a logic for shared mutable data structures. In *LICS*, 2002.
[18] Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames. In *Workshop on Formal Techniques for Java-like Programs*, 2008.
[19] Hongseok Yang, Oukseh Lee, Cristiano Calcagno, Dino Distefano, and Peter O'Hearn. Scalable shape analysis for systems code. In *CAV*, 2008.
[20] Karen Zee, Viktor Kuncak, and Martin Rinard. Full functional verification of linked data structures. In *PLDI*, 2008.

Probabilistic Aspects of Flash Filestores

Zoe Andrews[1], Annabelle McIver[2], Larissa Meinicke[2], and Carroll Morgan[3]

[1] School of Computing Science, Newcastle University, UK; Z.H.Andrews@ncl.ac.uk
[2] Department of Computing, Macquarie University, NSW 2109 Australia; {anabel, lmeinick}@ics.mq.edu.au
[3] School of Computer Science & Engineering, University of New South Wales, NSW 2052 Australia; carrollm@cse.unsw.edu.au

Abstract. Flash filestores have a variety of unique features that lead to interesting design constraints. One challenge is that of ensuring that each block of the drive is used and erased evenly, known as "wear-levelling". This paper presents a novel approach to the analysis of wear-levelling algorithms using probabilistic specification and analysis techniques. A simplified version of a real wear-levelling algorithm used in an actual flash filestore is given as an illustration. The expected lifetime of a flash filestore implementing such an algorithm is derived using probabilistic proof techniques.

1 Introduction

As part of the Grand Challenge on verified software [1], Joshi et al. proposed a mini-challenge on building a verifiable filesystem [2] using flash memory. The contribution of this paper (to the mini-challenge) is in the exploration of probabilistic aspects of flash filestores. The focus is on probabilistic "wear-levelling" algorithms (see Section 3), intended to maximise the life of flash memory. Probabilistic specification and analysis techniques are used to determine the expected lifetime of a flash filestore implementing a specific wear-levelling algorithm.

Since the mini-challenge was proposed, there have been several papers detailing their contribution to the problem. Butterfield and Woodcock [3] concentrated on a specific flash standard (ONFI) and developed a formal specification of it in Z. Kang and Jackson [4] modelled flash memory in Alloy, incorporating (very simplistic) wear-levelling and error recovery procedures. Damchoom et al. [5] focused on the data structure of the file system, decomposing an Event-B model into filestore and flash memory specific operations. The goal of Schierl et al. [6] was to understand the requirements of a "real system" and so they developed a formal model of such a system (UBIFS) from its code. The contribution of this paper is unique in that it explores *probabilistic* aspects of the flash filestore.

A brief introduction into probabilistic specifications can be found in Section 2, followed by (Section 3) an overview of why flash memory is interesting to examine and a definition of "wear-levelling". A probabilistic specification of a wear-levelling algorithm is presented in Section 4 and the expected lifetime of a flash filestore using such an algorithm is calculated. The results are discussed in Section 5 and some extensions are proposed. The paper concludes in Section 6.

	prog	*wp.prog.Q*
Assignment	$x := E$	$Q[x\backslash E]$
Composition	$prog_1; prog_2$	$wp.prog_1.(wp.prog_2.Q)$
Cond. choice	if G then $prog_1$ else $prog_2$ fi	$[G]\times wp.prog_1.Q + [\neg G]\times wp.prog_2.Q$
Probability	$prog_1\ {}_p\!\oplus\ prog_2$	$p*wp.prog_1.Q + (1-p)*wp.prog_2.Q$
While-loop	do $G \to body$ od	$(\mu X \quad [G]\times wp.body.X + [\neg G]\times Q)$

x is a program variable; E is an expression in the program variables; $prog_1$ and $prog_2$ are probabilistic programs; G is a Boolean-valued expression in the program variables; p is a constant probability in $[0, 1]$; and Q is an expectation (a real-valued expression in the program variables). Given an expression Q, we write $Q[x\backslash E]$ to mean expression Q in which free occurrences of x have been replaced by expression E. μ is the least fixed point operator w.r.t the ordering $\leq$ between expectations.

For expectations (interpreted as real-valued functions), scalar multiplication $*$, multiplication, $\times$, addition, $+$, subtraction, $-$, and the comparison (such as $\leq$ and $<$) between expectations are defined by the usual point-wise extension of these operators (as they apply to the real numbers). Multiplication and scalar multiplication have the highest precedence, followed by addition, subtraction, and finally the comparison operators. Operators of equal precedence are evaluated from the left.

$[\cdot]$ is the function that takes a Boolean expression *false* to 0 and *true* to 1. For $\{0, 1\}$ real-valued functions, operation $\leq$ means the same as implication over predicates, and $\times$ represents conjunction. Addition over disjoint predicates is equivalent to disjunction.

Fig. 1: Probabilistic program notation and weakest-precondition semantics.

2 Probabilistic specifications

To analyse probabilistic wear-levelling we use pGCL [7]. This is an extension of standard GCL [8] to include probabilistic choice (see Figure 1). Like GCL, it is a formalism that allows source-level reasoning about programs; it is a generalisation since it is able to handle *probabilistic* (as well as standard) properties.

There are (at least) three reasons for treating probabilistic properties as rigorously as standard properties: it allows

1. an accurate comparison of performance between differing designs;
2. the specification of basic average guarantees on performance together with design patterns which achieve them;
3. an exploration of the relationship between program parameters affecting overall performance.

Probabilistic pGCL introduces a probabilistic choice operator ${}_p\!\oplus$ (for $0 \leq p \leq 1$) to include the possibility of probabilistic updates. Thus $x := 1\ {}_p\!\oplus\ x := 2$, would mean that x is assigned the value 1 with probability p, and 2 otherwise (with probability $1-p$). With this, properties of interest are no longer necessarily absolute, but rather it is possible to reason about the probability that a property is established or, as shall be seen, *expected* or *average case* execution times.

Quantitative annotations are written over real-valued expressions of the state space, for example,

$$\{p\} \quad x := 1 \ {}_p\!\oplus \ x := 2 \quad \{[x = 1]\} \tag{1}$$

$$\{p + 2(1-p)\} \quad x := 1 \ {}_p\!\oplus \ x := 2 \quad \{x\} \tag{2}$$

The post-annotation is treated as a random variable[4] over the program variables. In (1) we use $[x = 1]$ for the characteristic random variable returning 0 or 1 depending on whether the condition "$x = 1$" is satisfied. In (2) the random variable is simply the value of x. The pre-annotation is the expected value of the random variable after execution of the program, thus for (1) it is p because the probabilistic update establishes "$x = 1$" with probability p; in (2) the pre-annotation is $p+2(1-p)$ because that is the expected value of x. More generally the pre-annotations will be sensitive to the initial state.

Formally an annotation is interpreted using the source-level *wp*-semantics set out in Figure 1, so that $\{P\}$ *prog* $\{Q\}$ is valid provided that $P \leq wp.prog.Q$.

The idea conveniently generalises loop "invariants" as follows. Consider the while-loop in Figure 1. A standard loop invariant, I, has to hold at the start of every iteration, and constrains the states that the loop can enter, formally written as $G \wedge I \Rightarrow wp.body.I$. A quantitative invariant, E, can also be defined for such a loop. The expected value of E cannot decrease throughout the loop, written $[G] \times E \leq \ wp.body.E$.

Like standard invariants, quantitative invariants can be used to reason about properties of the whole iteration. For example, for standard invariant I of the above loop we have that if the *loop* terminates (with probability 1) then $I \Rightarrow wp.loop.I$. Similarly we have that $E \leq wp.loop.E$ holds for quantitative invariant E if *loop* is certainly terminating.

Both standard and quantitative invariants are employed to reason about aspects of the flash filestore case study in Section 4.

3 Flash filestore systems

Flash memory is a popular storage medium for many applications due to its lack of moving parts. However, it also behaves differently to other storage media, e.g. magnetic disks, and new algorithms are required to deal with this new behaviour.

In particular, an individual bit stored on a flash memory cannot be overwritten; data has to be erased by block (between tens and hundreds of kilobytes) [9] before that space can be re-used. This is because individual bits can be cleared, but bits can be reliably set only a block at a time. Another important feature of flash memory is that each block can only be erased a fixed number (typically 10,000 to 1,000,000) [9] of times before becoming unreliable.

[4] A random variable is a function from the sample space to the real numbers, i.e. there is an element of chance in the value assigned to the variable.

Many different algorithms have been proposed [9] to deal with these characteristics. A number of these algorithms, many of them probabilistic, are designed to provide "wear-levelling" (see below). However, there has been little or no research into formally analysing the probabilistic aspects of these algorithms.

3.1 Wear-levelling

As each block of flash memory has a limited number of times it can be erased, it is important to ensure that individual blocks do not become worn out prematurely. For example if the same block was used and erased repeatedly a significant number of times it would no longer operate reliably and the storage capacity of the flash filestore would have to be reduced. The process of ensuring that the relative number of erasures for each block in the filestore remains approximately the same at all times is called *wear-levelling*.

When designing and evaluating wear-levelling algorithms, there are (at least) two conflicting characteristics to consider: its impact on the lifetime of the flash memory; and the speed at which it frees up space. A naïve algorithm, that guarantees that no block has been erased more than (say) c times more than any other block, may have a long lifetime (if c is small). However, it may also require a lot of relocation of data and therefore significantly affect the performance of the device. To overcome these issues a number of probabilistic algorithms have been proposed for wear-levelling. It is hoped that these algorithms will result in the blocks being worn evenly on average, so that the expected lifetime of the system remains long, but that they will have better performance characteristics than their non-probabilistic counterparts.

In this case study we investigate the probabilistic aspects of one particular probabilistic wear-levelling algorithm, taken from the JFFS flash file system [10]. For 99% of the time this algorithm selects a block for erasure to maximise the amount of memory that will be freed up; the other 1% of the time any block is chosen at random for reclamation. The idea is that blocks containing static valid data (which would be ignored 99% of the time) will eventually be chosen by the random selection and moved to a more worn block.

4 Specification of a flash filestore

Before considering any specific wear-levelling algorithms, let us semi-formally show what such an algorithm should do in the bigger picture of a flash filestore. In particular, the garbage collector functionality of a flash filestore is defined, as shown in Figure 2. It is assumed that the garbage collector is a continuous loop that runs alongside functionality to read to and write from the filestore. In every iteration the garbage collector selects two blocks: one to erase (A) and one to copy any valid data from A to.

The wear-levelling algorithm is the part of this process responsible for choosing the blocks to be reclaimed. The relevant lines of the garbage collector have been marked with $*$ in Figure 2. In section 4.1 these lines are expanded into

```
  GARBAGE COLLECTOR

* do true →
*    select a dirty block A;
     find another block B that has sufficient clean
       pages to receive the valid blocks from A;
     copy them over from A to B;
*    erase A;
* od
```

Fig. 2: Abstract representation of a flash filestore garbage collector

a formal description of a simple probabilistic wear-levelling algorithm based on that used in the JFFS [10] flash file system.

4.1 A probabilistic wear-levelling algorithm

In this section we show how to model a probabilistic wear-levelling algorithm using the language from Section 2. The algorithm chosen is a simplified version of that used in JFFS [10]. One simplification made was to restrict the number of blocks in the filestore to two. A second was to only include the probabilistic iterations of the algorithm in the model, i.e. the iterations that select a block according to the amount of space reclaimed have been omitted. We concentrate on the probabilistic part only as we believe that our approach is novel for analysing the probabilistic aspects of such algorithms. We restrict the number of blocks because we aim to illustrate how this kind of approach might work, whilst keeping the arithmetic as simple as possible.

The formal specification of the wear-levelling algorithm is shown in Figure 3. The variables m and n represent two different blocks and record the number of times that each has been erased so far. In each iteration of the algorithm one of these blocks is selected for reclamation: block m or block n, each with probability $\frac{1}{2}$. The total number of times both blocks have been erased, represented by variable e is also incremented each iteration. This variable is used to calculate the expected lifetime of the flash filestore in terms of number of erasures, more details to follow. Initially it is assumed that if either of the blocks reach some maximum number of erasures, N, then the flash filestore is retired.

4.2 Analysis

The probabilistic wear-levelling algorithm makes no guarantee that each block has been erased at most c (for $c < N$) times more than any other block. However, we can use the quantitative invariant

$$n - m$$

$$
\begin{array}{l}
\mathsf{do}\ \ m < N \wedge n < N \rightarrow \\
\quad\ \ m := m + 1 \,{}_{\frac{1}{2}}\!\oplus n := n + 1; \\
\quad\ \ e := e + 1 \\
\mathsf{od}
\end{array}
$$

- m represents the number of times block m has been erased, similarly for n
- e represents the total number of times any block has been erased
- N represents the maximum number of times a block can be erased

Fig. 3: Specification of the probabilistic wear-levelling algorithm

to show that the average difference between the number of erasures of each block is zero (after each execution of the body of the loop). Intuitively, this property holds because of the symmetry of the probabilistic choice.

But how does this wear-levelling characteristic affect the lifetime of the flash filestore? We now analyse the wear-levelling algorithm above to determine the expected lifetime it provides. We chose to measure the lifetime of the device in terms of the number of erasures. This involved determining the expected value of the (random) variable e on termination of the loop, written $E[e]$, which is calculated with the assistance of loop invariants (Section 2).

Using the standard invariant

$$e = m + n$$

we have that $E[e] = E[m+n]$ and so it is enough to calculate the expected value of $m + n$; but to do this we need a more complicated invariant. Knowledge of the negative binomial probability distribution provides us with one.

Recall [11] that the negative binomial distribution models the number of trials required until x instances of a specific event have been observed, assuming that the probability of x occurring is constant across all trials. Consider the expected value of m alone initially: this can be thought of as the event of interest for a negative binomial distribution, with N being the target number of instances required. However, this distribution allows situations not permitted by our model, for example the case where $m = N$ and $n = N + 1$. It is necessary to include the situation where n reaches N before m does. To resolve this the negative binomial distribution is adapted to have an upper bound on the number of trials allowed and have two instances of the distribution – one each for the situations in which m and n reach N first. Using the basis of the negative binomial distribution, and incorporating the complications described above, it turns out that there exists another quantitative invariant of the loop, as shown in Equation 3. Intuitively this equation assumes that $m + n$ trials have already occurred (first line) and calculates the expected number of remaining trials given this fact (second line).

$$m+n+\sum_{e=N-m}^{2N-(m+n+1)} e\binom{e-1}{N-m-1}\left(\frac{1}{2}\right)^e + \sum_{e=N-n}^{2N-(m+n+1)} e\binom{e-1}{N-n-1}\left(\frac{1}{2}\right)^e \quad (3)$$

It can be formally confirmed that the expression (call it *inv*) shown in Equation 3 is a quantitative invariant of the loop. This requires (Section 2) showing that $[G] \times inv \leq wp.body.inv$. Using the *wp* proof rules defined in Section 2 it can be shown that $wp.body.inv \equiv inv$ as is summarised below. It can also be trivially proved that the values of m and n on termination are as expected (e.g. that $inv\,(N,n,e) = N+n$).

$$\begin{array}{ll}
& wp.body.inv \\
\equiv & wp.\left(\left(m := m+1 \;{}_{\frac{1}{2}}\!\oplus\; n := n+1\right);\; e := e+1\right).inv \quad \text{definition of } body \\
\equiv & \quad \text{composition, assignment, probability and definition of } inv \\
& \frac{1}{2} * \left(\begin{array}{l} m+1+n+ \displaystyle\sum_{e=N-(m+1)}^{2N-((m+1)+n+1)} e\binom{e-1}{N-(m+1)-1}\left(\frac{1}{2}\right)^e \\ + \displaystyle\sum_{e=N-n}^{2N-((m+1)+n+1)} e\binom{e-1}{N-n-1}\left(\frac{1}{2}\right)^e \end{array} \right) \\
& + \frac{1}{2} * \left(\begin{array}{l} m+n+1+ \displaystyle\sum_{e=N-m}^{2N-(m+(n+1)+1)} e\binom{e-1}{N-m-1}\left(\frac{1}{2}\right)^e \\ + \displaystyle\sum_{e=N-(n+1)}^{2N-(m+(n+1)+1)} e\binom{e-1}{N-(n+1)-1}\left(\frac{1}{2}\right)^e \end{array} \right) \\
\equiv & \quad \text{simple algebra including combination and summation rules} \\
& m+n+ \displaystyle\sum_{e=N-m}^{2N-(m+n+1)} e\binom{e-1}{N-m-1}\left(\frac{1}{2}\right)^e \\
& \quad + \displaystyle\sum_{e=N-n}^{2N-(m+n+1)} e\binom{e-1}{N-n-1}\left(\frac{1}{2}\right)^e \\
\equiv & inv \quad \text{definition of } inv
\end{array}$$

The expected lifetime of the flash filestore ($E[e]$) can be determined from *inv* by substituting m and n with their initial values (both 0). This gives the expression found in Equation 4.

$$E[e] = 2 * \sum_{e=N}^{2N-1} e\binom{e-1}{N-1}\left(\frac{1}{2}\right)^e \quad (4)$$

Using the expression in Equation 4, the expected lifetime of the flash filestore was calculated for various values of N, tabulated in Figure 4. It can be seen

N	E[e]	E[e]/N
2	2.50	1.25
3	4.13	1.38
4	5.81	1.45
5	7.54	1.51
10	16.48	1.65
50	92.04	1.84
100	188.73	1.89
500	974.77	1.95
1000	1964.32	1.96

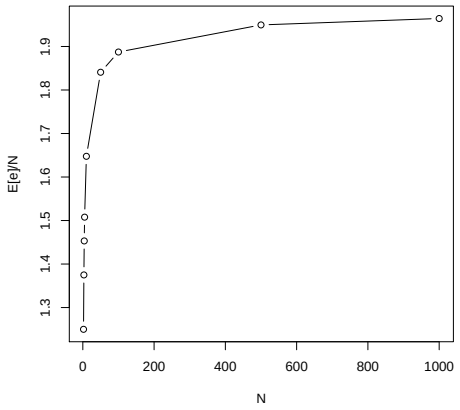


Fig. 4: Expected lifetime ($E[e]$) for the abstract wear-levelling algorithm

that $E[e]/N$ approaches two (Figure 4). Note that an algorithm that alternated between blocks (i.e. ensured each block had been erased at most once more than the other) would have an expected lifetime of $2N - 1$ for any N. However, it may have worse performance characteristics.

5 Discussion

We have illustrated how probabilistic analysis can be used to determine the expected lifetime of a flash filestore for a simple wear-levelling algorithm. We discuss this result in more detail here. In particular, we discuss what it means for a flash filestore to fail and require replacement and then consider possible extensions to this research.

5.1 When should the filestore be retired?

There are several options available for deciding when a flash filestore is no longer useful and retiring it. They are discussed in turn below.

A block reaches its maximum number of erasures. One option is to retire the flash filestore as soon as a block reaches the maximum erasures allowed. This seems rather extreme, but may be necessary for applications that require a high percentage of the total disk space to operate correctly. If such a strict measure is not required the affected block could be marked as unusable and the remaining blocks of the filestore continue to be used until one of the following cases occurs.

Valid data can not be relocated. Another strategy could be to continue until an erase procedure is not possible due to insufficient free space on the

filestore. When a block is erased, it is necessary to find sufficient free space to move the valid data remaining on the chosen block to. If sufficient space can not be found regardless of the block chosen for erasure, the filestore should be retired immediately otherwise data will be lost.

Data can not be written. Alongside the garbage collector there is also a process running to handle the read and write requests on the filestore. It may be necessary to retire the filestore if a write is requested and insufficient free space exists for the write. This strategy would be needed in situations where the incoming data rate is high and there is a limited buffer in which data pending a write operation can be stored.

The algorithm modelled in Section 4 uses the first of these approaches, as the loop terminates when either of the blocks reaches the maximum value. Using the second retirement criteria *may* extend the expected lifetime of this filestore. However, it would only be possible if (once a block has reached its maximum number of erasures) the remaining block has no valid data when the next erasure occurs. This depends on the schedule of writes and erasures for the application it is being used for. The final approach could either increase or decrease the expected lifetime of the filestore, it depends as much on the rate of write operations and write buffer size as it does on the choice of wear-levelling algorithm.

5.2 Possible extensions

Modelling and analysing alternative retirement strategies as discussed above is a challenging extension to the research because it requires the addition of read, write and erase rates and durations to the model. However, if such data were added it may also be possible to analyse the trade-off between the expected lifetime of the flash filestore and the performance issues mentioned in Section 3. Such analysis is complex and would benefit from the formal modelling of continuous probability distributions, which is still ongoing research [12].

Finally, there are lots of proposed algorithms for wear-levelling [9], this research could be extended by analysing and comparing a variety of these.

6 Conclusions

This work has been motivated by the Grand Challenge on verified software [1], in particular the mini-challenge on a verified filestore [2]. The contribution of this paper to the mini-challenge has been to examine probabilistic aspects of flash filestores. More specifically, an insight has been provided into how probabilistic wear-levelling algorithms for flash memory can be analysed using probabilistic specification and proof techniques. Using this approach it has been shown how the expected lifetime of a flash filestore implementing a simplified version of JFFS flash [10] can be found using *quantitative invariants*.

The results of the analysis have been discussed and different termination scenarios have been considered depending on the intended usage of the flash filestore. Finally some extensions to the research have been suggested.

This research has produced some useful results, however, it has also highlighted the complexity of finding suitable invariants for probabilistic specifications. Katoen et al. are currently working on techniques for finding quantitative invariants [13], which would assist in analysing the more complex specifications required to model more realistic systems. It would be interesting to re-visit this case study when such research matures.

Acknowledgements

We are grateful to John Fitzgerald, Rajeev Joshi, Jim Woodcock, Michael Butler and Kriangsak Damchoom for their valuable discussions on this work. Many useful collaborations would not have been possible without the support of the IST FP7 DEPLOY project. The research was funded by the Australian Research Council Grant DP0879529 and a UK EPSRC Ph.D. stipend. We are grateful for the financial support provided by the School of Computer Science and Engineering at the University of New South Wales, which allowed Andrews to visit and work in Australia under UNSW's Practicum Exchange Program.

References

1. Jones, C., O'Hearn, P., Woodcock, J.: Verified software: A grand challenge. Computer **39** (2006) 93–95
2. Joshi, R., Holzmann, G.J.: A mini challenge: build a verifiable filesystem. Form. Asp. Comput. **19**(2) (2007) 269–272
3. Butterfield, A., Freitas, L., Woodcock, J.: Mechanising a formal model of flash memory. Sci. Comput. Program. **74**(4) (2009) 219–237
4. Kang, E., Jackson, D.: Formal modeling and analysis of a flash filesystem in Alloy. In: ABZ '08: Proceedings of the 1st international conference on Abstract State Machines, B and Z, Berlin, Heidelberg, Springer-Verlag (2008) 294–308
5. Damchoom, K., Butler, M.: Applying event and machine decomposition to a flash-based filestore in Event-B, Berlin, Heidelberg, Springer-Verlag (2009) 134–152
6. Schierl, A., Schellhorn, G., Haneberg, D., Reif, W.: Abstract specification of the UBIFS file system for flash memory. In: FM '09: Proceedings of the 2nd World Congress on Formal Methods, Berlin, Heidelberg, Springer-Verlag (2009) 190–206
7. McIver, A., Morgan, C.: Abstraction, Refinement And Proof For Probabilistic Systems (Monographs in Computer Science). Springer (2004)
8. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall, N.J. (1976)
9. Gal, E., Toledo, S.: Algorithms and data structures for flash memories. ACM Comput. Surv. **37**(2) (2005) 138–163
10. Woodhouse, D.: JFFS: The Journalling Flash File System. In: Proceedings Ottawa Linux Symposium. (2001)
11. Rice, J.A.: Mathematical Statistics and Data Analysis. Duxbury Press (2001)
12. Andrews, Z.: Towards a stochastic Event-B for designing dependable systems. Technical Report CS-TR-1154, Newcastle Univ., School of Comp. Sci. (July 2009)
13. Katoen, J., McIver, A., Meinicke, L., Morgan, C.: Linear-invariant generation for probabilistic programs: automated support for proof-based methods. In: To appear in The Seventeenth International Static Analysis Symposium (SAS 2010). (2010)

Functional Correctness for Pointer Programs

Ewen Maclean[1], Andrew Ireland[1], and Gudmund Grov[2]

[1] School of Mathematical and Computer Sciences, Heriot-Watt University, Edinburgh, {E.A.H.Maclean,A.Ireland}@macs.hw.ac.uk
[2] School of Informatics, University of Edinburgh, ggrov@inf.ed.ac.uk

Abstract. We introduce the CORE system which builds on existing systems, such as Smallfoot [3], to automatically prove functional correctness of programs which manipulate pointers, in some cases generating missing parts of loop-invariants. We describe novel techniques which can be generalised to apply to more complicated data structures.

1 Motivation

We present here work on proving correct programs which reason about pointers. We describe the background to the work, and explain how we build on tools such as Smallfoot [3], extending them to reason about full functional correctness, and synthesising missing parts of loop-invariants in some cases.

1.1 Separation logic

Separation logic was developed as an extension to Hoare logic [8], with the aim of simplifying pointer program verification proofs [14, 15]. Pointers are a powerful and widely used programming mechanism, but developing and maintaining correct pointer programs is notoriously hard. A key feature of separation logic is that it focuses the reasoning effort on only those parts of the heap that are relevant to a program, so called *local reasoning*. Because it deals smoothly with pointers, including "dirty" features such as memory disposal and address arithmetic, separation logic holds the promise of allowing verification technology to be applied to a much wider range of **real-world software** than has been possible up to now.

In terms of tool development, the main focus has been on shape analysis. Such analysis can be used to verify properties about the structure (*shape*) of data structures within the heap. In this paper we extend this analysis to the verification of functional properties.

1.2 Proof planning

Proof planning is a technique for automating the search for proofs through the use of high-level proof outlines, known as *proof plans* [4]. The current state-of-the-art proof planner is called IsaPlanner [7], which is Isabelle based. Proof planning

has been used extensively for proof by mathematical induction [6]. Mathematical induction is essential for the synthesis and verification of the inductively defined predicates that arise within separation logic specifications. Proof planning therefore offers significant benefits for reasoning about separation logic specifications. Proof planning techniques have been applied successfully to the problems of inductive conjecture generalisation and lemma discovery [9, 10], as well as loop invariant discovery [12]. This work is currently being integrated and extended within IsaPlanner. The tool integration capabilities of proof planning have been demonstrated through the Clam-HOL [16] and NuSPADE projects[3] [11]. The NuSPADE project targeted the SPARK Approach [2], and integrated proof planning with light-weight program analysis in order to increase proof automation for loop-based code. The resulting integration was applied to industrial strength problems and successfully increased the level of proof automation for exception freedom proofs [11].

2 Methodology

In this section we describe the methodology of our system. We make forward references to the syntax of Smallfoot$^+$and to the program which becomes the focus of our investigation.

2.1 Shape, structure and function

The division between shape and function is well documented in the separation logic literature, but in order to elucidate our proof-techniques we introduce the notion of "Structural Content". The structural part can be seen as the "glue" between shape and function. In what follows, $data\_lseg(\alpha, i, j)$ is a linked list pointed to by i, and terminated by j whose data elements are represented by the functional list α. $data\_list(\alpha, i)$ is equivalent to $data\_lseg(\alpha, i, nil)$. In the loop invariant expression

$$data\_lseg(\alpha, i, nil) * data\_lseg(\beta, j, nil) \wedge \alpha_o = rev(\beta) <> \alpha$$

The three categories are defined as

Shape This describes purely the shape of the heap, and hence can be described purely as in Smallfoot as

$$list(i) * list(j).$$

as the list segments are null-terminated. There is no information about any data that is contained in the list, purely an indication of the inductive data structures that exist in this part of the heap, in this case linked lists.

[3] NuSPADE project: http://www.macs.hw.ac.uk/nuspade

Structural This describes the inductive structures on the heap, and attributes names for the data contained within them. In this case the structural content is written

$$data\_list(\alpha, i) * data\_list(\beta, j)$$

in our system, as the list segments are null-terminated. When reasoning about functional properties, it is important that shape information is augmented with logical variables so that this can be extracted.

Functional This describes the *pure* fragment of the statement in separation logic. In this case

$$\alpha_0 = rev(\beta) <> \alpha$$

describes the functional content, which importantly relies on the logical variables introduced between the shape and structural content.

2.2 Extracting Verification Conditions

The CORE system builds on a verified verification condition generator [1], which guarantees the correctness of the proof obligations on which to work. In order to investigate functional properties we extend the language of Smallfoot to Smallfoot$^+$. Any meta-variables introduced into the assertions are written in capital letters, as described in section §3.2.

2.3 Proof Techniques

In order to show how we separate functional content from shape content we must first describe the rewrite-rule system we use. Two central rewrite-rules we employ are

$$A \mathrel{-\!\!*} (A * B) \Rightarrow B \tag{1}$$

$$i \neq j \rightarrow data\_lseg(\alpha, i, j) \Rightarrow \exists \alpha_h, \alpha_t, k.[i \overset{data}{\mapsto} \alpha_h] * [i \overset{next}{\mapsto} k] * data\_lseg(\alpha_t, k, j) \wedge \alpha = \alpha_h :: \alpha_t \tag{2}$$

which when applied to the goal, $A \Rightarrow B$ rewrites according to a rule which corresponds to $B \rightarrow A$, where $\rightarrow$ is logical implication.

Rule (1) is central to proof in separation logic. The extension to the Hoare logic rules introduce a $\mathrel{-\!\!*}$ symbol at every assignment or allocation of a pointer. In order to complete a proof, we aim to have no such symbols remaining, meaning that we are able to determine the shape of the concrete heap in the hypotheses and conclusions. The only existentials that remain should pertain to functionality. Rule (2) describes the way in which a list segment can be "decomposed". In this case, if the list segment is not empty, the leading cons cell is isolated from the front of the list.

Mutation Mutation is a technique developed in the CORE system to mirror some of the initial motivations of *rippling* [5]. It is not as formal, as it has no calculus, and no termination argument, but is capable of generating sets of rewrite-rules.

Brief Example In order to apply rule (1) we must extract the precedent to $-\!\!*$ from the heap in the postcedent. For example in the case where we have a conclusion to manipulate

$$i \overset{data}{\mapsto} y * (i \overset{next}{\mapsto} z -\!\!* data\_lseg(\alpha, i, nil))$$

we need to extract from the list a cons cell from the list segment which corresponds to the precedent. In this case we can exploit rule (2) and bring a matching cons cell from the list segment to the from the front, and apply rule (1), and the associative-commutative properties of $*$ to yield

$$i \overset{data}{\mapsto} y * i \overset{next}{\mapsto} z -\!\!* (\exists \alpha_h, \alpha_t . i \overset{data}{\mapsto} \alpha_h * i \overset{next}{\mapsto} j * data\_lseg(\alpha_t, j, nil)$$

at which point rule (1) applies leaving the goal

$$\exists \alpha_h, \alpha_t . i \overset{data}{\mapsto} y * (i \overset{data}{\mapsto} \alpha_h * data\_lseg(\alpha_t, j, nil))$$

Note that in this case we need to prove that $x \neq nil$.

General Case In general, as we are dealing with pointer programs whose verification conditions are generated using the extended Hoare logic described in [15], the precedent to $-\!\!*$ will always be of the form $[x \overset{next}{\mapsto} y]$ where x is always a variable, and y can be existentially quanitified. We are focusing in this presentation on linked lists, but our analysis can apply to any inductive data structure which uses pointers.

We describe here the possible situations when decomposing a linked list in order to match the precedent $[x \overset{next}{\mapsto} y]$. We assume in what follows that $\mathcal{F}_i$ is an existentially quantified variable, and x, y, z are ground varibles. Attached to each possibility is a score, which represents the number of existential variables introduced, which will be explained in the following section.

score= 10 $[x \overset{next}{\mapsto} y]$
This case represents an exact ground match.

score= 6 $[x \overset{next}{\mapsto} \mathcal{F}_1]$
This case represents an inexact match, and instantiates the $\mathcal{F}_1$ with y.

score= 3 $data\_lseg(\alpha, x, \mathcal{F}_1)$
This case represents a match with the first cons of the linked list. This generalises to other data structures where the "leading" pointer can be taken from the front of the inductive structure.

score= 3 $data\_lseg(\alpha, \mathcal{F}_1, y)$
This case represents a match with the last cons of the linked list. This generalises to other data structures where the "trailing" pointer can be taken from the back of the inductive structure.

score= 1 $data\_lseg(\alpha, \mathcal{F}_1, \mathcal{F}_2)$
This case represents a match with the a cons somewhere embedded within the linked list.

Terminating Associative-Commutative Rewriting When mutating, a candidate pointer of the form $[x \overset{next}{\mapsto} y]$ is "pulled" to the front of an expression representing a heap, in order that rule $-\!\!*$ can apply. To do this we use rules such as

$$X * Y \Rightarrow Y * X \tag{3}$$

$$pure(Z) \rightarrow X * (Y \wedge X) \Rightarrow Y * (X \wedge Z) \tag{4}$$

$$pure(Z) \rightarrow (X \wedge Z) * Y \Rightarrow X * (Z \wedge Y) \tag{5}$$

$$(X * Y) * Z \Rightarrow X * (Y * Z) \tag{6}$$

where *pure* is a predicate which denotes that its argument has no shape content.

The CORE system identifies the position of a term in the term tree of the postcedent to $-\!\!*$. For example in term $a * (b * (\mathcal{X} \wedge b))$, the term $\mathcal{X}$ is at position $[2, 2, 1]$. Given an identifiable and sufficient set of rewrite rules as shown above, the heap can be mutated so that the term $\mathcal{X}$ is at position [1]. This process is done in the two stages, depicted in Figure 1. This is a terminating process since the number of occurrences of a node labelled with $n > 1$ is decreasing in the first two rewrite rules, and the length of the position in the tree is decreasing in the subsequent rule applications [1]. The reason why a list of rewrite rules and positions is generated is so that a tactic can be generated, and sent to a theorem prover such as Coq or Isabelle using an implementation such as that described in [17].

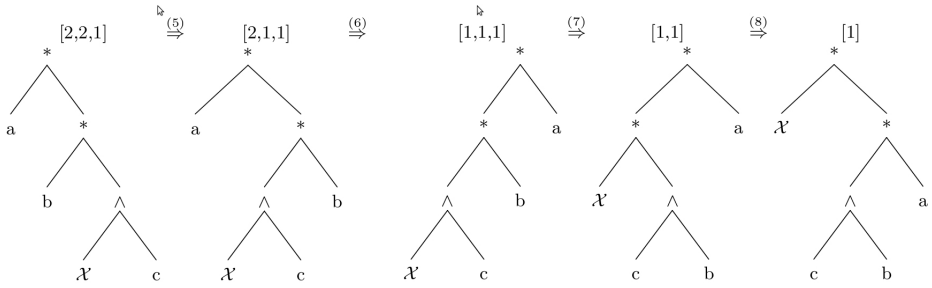


Fig. 1. Moving the desired term to the top of the tree

Structural Fertilisation and Functional Residue Once mutation has been exhaustively applied, we are left with a heap in the hypothesis and a heap in the conclusion. The job now is to match the two heaps and extract the functional residue. For example if we have a goal

$$\frac{[x \overset{next}{\mapsto} \mathcal{X}_1] * [x \overset{data}{\mapsto} \mathcal{X}_2] * data\_lseg(\mathcal{X}_3, y, nil) \wedge \mathcal{X}_2 :: \mathcal{X}_3 = \mathcal{X}_4}{data\_lseg(\mathcal{F}_1, y, nil) * [x \overset{next}{\mapsto} \mathcal{F}_2] * [x \overset{data}{\mapsto} \mathcal{F}_3] \wedge \mathcal{X}_4 = append(\mathcal{F}_3 :: nil, \mathcal{F}_1)}$$

where the $\mathcal{X}_i$ are constants, and the $\mathcal{F}_i$ are Skolem functions (with their dependencies omitted). Observing this simple example, we can see that the shape and structural elements of the heap can be made to match with the substitutions

$$\mathcal{X}_3/\mathcal{F}_1 \qquad \mathcal{X}_1/\mathcal{F}_2 \qquad \mathcal{X}_2/\mathcal{F}_3$$

This leaves a functional "residue" which is a goal that must be proved. In this case we need to prove the simple goal

$$\vdash \mathcal{X}_2 :: \mathcal{X}_3 = append(\mathcal{X}_2 :: nil, \mathcal{X}_3).$$

Term Synthesis and Counter-example Checking We introduce a system whereby if a loop-invariant is incomplete in some regard, we synthesise terms to a certain depth, and use a counter-example checker to filter out incorrect theorems. This at present relies on a theory file for lists, but we intend to automate a process whereby the possible functions over which to range can be automatically generated. In the case of lists we assume symbols

$$rev \qquad append \qquad nil \qquad cons \qquad = \qquad \wedge \qquad \neg \qquad true$$

which allows us to construct conjectures about lists. For an example of this see §3.2. Incorrect conjectures are ruled out using a simple implemented counter example checker which checks lists populated with lists of integers.

3 Example

In this section we present in detail the verification of a canonical test for separation logic – the in-place reversal of a list, whose shape version has been proved automatically in work such as that described in [13]. We present here an automatic proof of a fully functional version of the program, shown with annotations in Figure 2. Initially we show how the proof proceeds, using which proof assistants, given a loop invariant, and then go on to describe a synthesis story where only the shape part of the loop invariant is given.

3.1 Verification story

With a specified loop invariant, the CORE tool is able to automatically separate the shape and functional parts of the verification conditions, and send the resulting functional residue to IsaPlanner. In the following presentations of the proofs, the sequents are generated automatically by the system, but for ease of reading, have been slightly modified by hand. For the purposes of the description of verification, we will concentrate on the proof that the invariant holds.

The system first inserts Skolem functions and constants for the quantified variables. For ease of presentation we omit the variable dependencies from the

```
hd,tl;

list_reverse(o;i) [data_list(a;i)] {
  local t;
  o = NULL;
  while (i != NULL) [Ex alpha. Ex beta. data_list(alpha;i) *
    data_list(beta;o) /\ a = append(reverse(beta),alpha)] {
    t = i->tl;
    i->tl = o;
    o = i;
    i = t;
  }
} [data_list(reverse(a);o)]
```

Fig. 2. In-place list reversal

Skolem functions:

$$\frac{(l1 \neq null \wedge (lseg(\mathcal{X}_1, l1, null) * (lseg(\mathcal{X}_2, l0, null) \wedge \mathcal{X}_a = append(reverse(\mathcal{X}_2), \mathcal{X}_1))))}{\begin{array}{l}([l1 \overset{next}{\mapsto} \mathcal{F}_1] * ([l1 \overset{next}{\mapsto} \mathcal{F}_1] \mathrel{-\!\!*} \\ ([l1 \overset{next}{\mapsto} \mathcal{F}_4] * ([l1 \overset{next}{\mapsto} l0] \mathrel{-\!\!*} \\ (lseg(\mathcal{F}_2, \mathcal{F}_1, null) * (lseg(\mathcal{F}_3, l1, null) \wedge \mathcal{X}_a = append(reverse(\mathcal{F}_3), \mathcal{F}_2)))))))\end{array}}$$

At this point in the proof, the central analysis procedure can be employed. In order to complete the proof we need to eliminate the $\mathrel{-\!\!*}$ symbols. We exploit the central rewrite-rule

$$A \mathrel{-\!\!*} (A * B) \Rightarrow B$$

In order to do this we need to decide on instantiations for existential variables – now Skolem functions denoted by $\mathcal{F}_n$. As described in §2.3, the system now determines the best way in which to decompose the postcedent to the structural implication $\mathcal{F}_n$. In this case the possible elements which can unify with the precedent $[l1 \overset{next}{\mapsto} l0]$, with the given scores, are

Score 1 – $lseg(\mathcal{F}_2, \mathcal{F}_1, null)$ This list segment can be decomposed by taking the pointer $\mathcal{F}_1$ from the end of the list. This introduces new existentials for the data, and instantiates the existential $\mathcal{F}_1$

Score 2 – $lseg(\mathcal{F}_3, l1, null)$ This list segment can be decomposed by taking the pointer $l1$ from the end of the list. This is scored higher because there is an exact match with the variables.

Choosing the higher scored version, this now rewrites to

$$\frac{(l1 \neq null \wedge (lseg(\mathcal{X}_1, l1, null) * (lseg(\mathcal{X}_2, l0, null) \wedge \mathcal{X}_a = append(reverse(\mathcal{X}_2), \mathcal{X}_1))}{\begin{array}{l}([l1 \overset{next}{\mapsto} \mathcal{F}_1] * ([l1 \overset{next}{\mapsto} \mathcal{F}_1] \mathrel{-\!\!*} \\ ([l1 \overset{next}{\mapsto} \mathcal{F}_4] * ([l1 \overset{next}{\mapsto} l0] \mathrel{-\!\!*} \\ ((\exists x1.(\exists x2.(\exists x3.(([l1 \overset{data}{\mapsto} x2] * ([l1 \overset{next}{\mapsto} x3] * lseg(x1, x3, null)) \wedge cons(x2, x1) = \mathcal{F}_3) * \\ (\mathcal{X}_a = append(reverse(\mathcal{F}_3), \mathcal{F}_2) \wedge lseg(\mathcal{F}_2, \mathcal{F}_1, null)))\end{array}}$$

We omit the intermediate proof steps here, but after applying heuristically the techniques shown in §2.3, we end up with the entailment

$$\frac{(l1 \neq null \wedge ((([l1 \overset{data}{\mapsto} \mathcal{X}_4] * ([l1 \overset{next}{\mapsto} \mathcal{X}_5] * lseg(\mathcal{X}_3, \mathcal{X}_5, null)) \wedge cons(\mathcal{X}_4, \mathcal{X}_3) = \mathcal{X}_1 * \\ (lseg(\mathcal{X}_2, l0, null) \wedge \mathcal{X}_a = append(reverse(\mathcal{X}_1), \mathcal{X}_2))}{([l1 \overset{next}{\mapsto} \mathcal{F}_1] * (lseg(\mathcal{F}_5, l0, null) * ([l1 \overset{data}{\mapsto} \mathcal{F}_6] * (cons(\mathcal{F}_6, \mathcal{F}_5) = \mathcal{F}_3 \wedge \\ (\mathcal{X}_a = append(reverse(\mathcal{F}_2), \mathcal{F}_3) \wedge lseg(\mathcal{F}_2, \mathcal{F}_1, null)}$$

The instantiations calculated are

$$\{\mathcal{X}_4/\mathcal{F}_6, \mathcal{X}_5/\mathcal{F}_1, \mathcal{X}_2/\mathcal{F}_5, \mathcal{X}_3/\mathcal{F}_2\}$$

This leaves functional residue:

$$\frac{l1 \neq null \wedge cons(\mathcal{X}_4, \mathcal{X}_3) = \mathcal{X}_1 \wedge \mathcal{X}_a = append(reverse(\mathcal{X}_2), \mathcal{X}_1)}{cons(\mathcal{X}_4, \mathcal{X}_2) = \mathcal{F}_3 \wedge \mathcal{X}_a = append(reverse(\mathcal{F}_3), \mathcal{X}_3)}$$

which gets simplified to

$$l1 \neq null \vdash append(reverse(cons(\mathcal{X}_4, \mathcal{X}_3)), \mathcal{X}_2) = append(reverse(\mathcal{X}_3), cons(\mathcal{X}_4, \mathcal{X}_2))$$

which is easily proved by IsaPlanner.

3.2 Synthesis story

The program shown in Figure 2 includes a fully specified loop invariant. Imagine the same program with an incomplete loop invariant, where we introduce the meta-variable P, which is an unknown predicate, representing the functional part of the loop-invariant:

```
[Ex alpha. Ex beta. data_list(alpha;i) * data_list(beta;o) /\ P[a,alpha,beta]]
```

The functional residue is

$$\frac{l1 \neq null \wedge P[\mathcal{X}_a, cons(\mathcal{X}_4, \mathcal{X}_3), \mathcal{X}_2]}{P[\mathcal{X}_a, \mathcal{X}_3, cons(\mathcal{X}_4, \mathcal{X}_2)]}$$

for which we synthesise term

$$P \equiv \lambda x, y, z.\ x = append(rev(z), y)$$

4 Tool Integration Perspective

Figure 3 shows the tool chain of the program. Labels with **bold** font indicates that this is a topic and contribution of this paper. A stippled line indicates that this has not been integrated in the tool chain yet. ▭ represents a process, while ▯ indicates a process which is a "glue" between representations.

An annotated Smallfoot$^+$ program, which is a Smallfoot program where the annotations are extended with structural and functional properties is the input of the tool. This programs is fanned out to two processes: **shape filter** which

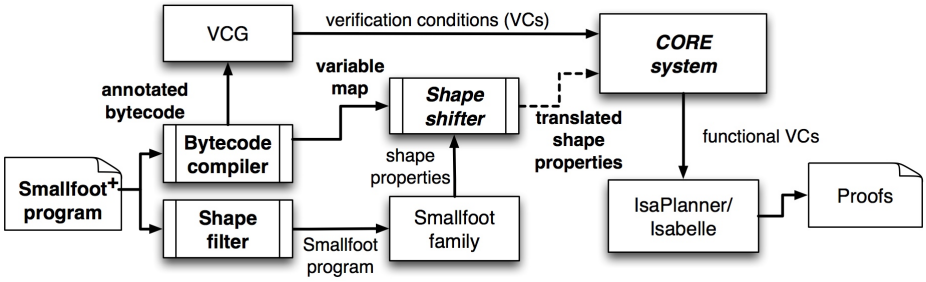


Fig. 3. The tool chain.

generates Smallfoot program, which is then sent to the Smallfoot family of tools for shape analysis; and a **bytecode compiler** which generates annotated bytecode, and well as a variable map. The annotated bytecode is fed into Atkey's sound verification condition generator [1], and the generated VCs is the input of the **CORE system** – the heart of the toolchain. By using the shape properties from the Smallfoot family of tools, the CORE system extracts purely functional properties properties which Isabelle/IsaPlanner [7] is used to verify.

Thus, our tool chain integrates 4 tools or family of tools: Atkey's sound VC generator; the Smallfoot family of tools; Isabelle and IsaPlanner; and our CORE planner. In addition, there are tools to translate between representations.

5 Conclusion

We have introduced a system, still under development, which aims to automatically prove the correctness of functional specifications about programs which reason about pointers. We believe that the techniques such as mutation and term synthesis which we have described augment existing techniques and can be generalised to apply to programs which involve more complicated inductive data structures.

References

1. Robert Atkey. Amortised resource analysis with separation logic. In Andrew D. Gordon, editor, *ESOP*, volume 6012 of *Lecture Notes in Computer Science*, pages 85–103. Springer, 2010.
2. J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley, 2003.
3. J. Berdine, C. Calcagno, and P. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, volume 4111 of *Lecture Notes in Computer Science*, pages 115–137. Springer, 2005.
4. A. Bundy. The use of explicit plans to guide inductive proofs. In R. Lusk and R. Overbeek, editors, *9th International Conference on Automated Deduction*, pages

111–120. Springer-Verlag, 1988. Longer version available from Edinburgh as DAI Research Paper No. 349.
5. A. Bundy, D. Basin, D. Hutter, and A. Ireland. *Rippling: Meta-level Guidance for Mathematical Reasoning*. Cambridge University Press, 2005.
6. A. Bundy, F. van Harmelen, J. Hesketh, and A. Smaill. Experiments with proof plans for induction. *Journal of Automated Reasoning*, 7:303–324, 1991. Earlier version available from Edinburgh as DAI Research Paper No 413.
7. L. Dixon and J. D. Fleuriot. IsaPlanner: A prototype proof planner in Isabelle. In *Proceedings of CADE'03*, volume 2741 of *LNCS*, pages 279–283, 2003.
8. C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–583, 1969.
9. A. Ireland and A. Bundy. Productive use of failure in inductive proof. *Journal of Automated Reasoning*, 16(1–2):79–111, 1996. Also available as DAI Research Paper No 716, Dept. of Artificial Intelligence, Edinburgh.
10. A. Ireland and A. Bundy. Automatic verification of functions with accumulating parameters. *Journal of Functional Programming: Special Issue on Theorem Proving & Functional Programming*, 9(2):225–245, March 1999. A longer version is available from Dept. of Computing and Electrical Engineering, Heriot-Watt University, Research Memo RM/97/11.
11. A. Ireland, B. J. Ellis, A. Cook, R. Chapman, and J. Barnes. An integrated approach to high integrity software verification. *Journal of Automated Reasoning: Special Issue on Empirically Successful Automated Reasoning*, 36(4):379–410, 2006.
12. A. Ireland and J. Stark. Proof planning for strategy development. *Annals of Mathematics and Artificial Intelligence*, 29(1-4):65–97, February 2001. An earlier version is available as Research Memo RM/00/3, Dept. of Computing and Electrical Engineering, Heriot-Watt University.
13. Roman Manevich, Eran Yahav, Ganesan Ramalingam, and Shmuel Sagiv. Predicate abstraction and canonical abstraction for singly-linked lists. In Radhia Cousot, editor, *VMCAI*, volume 3385 of *Lecture Notes in Computer Science*, pages 181–198. Springer, 2005.
14. P. O'Hearn, J. Reynolds, and Y. Hongseok. Local reasoning about programs that alter data structures. In *Proceedings of CSL'01*, volume 2142 of *Lecture Notes in Computer Science*, Paris, 2001.
15. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science*, pages 55–74. IEEE Computer Society, 2002.
16. K. Slind, M. Gordon, R. Boulton, and A. Bundy. System description: An interface between CLAM and HOL. In C. Kirchner and H. Kirchner, editors, *15th International Conference on Automated Deduction*, volume 1421 of *Lecture Notes in Artificial Intelligence*, pages 134–138, Lindau, Germany, July 1998. Springer. Earlier version available from Edinburgh as DAI Research Paper 885.
17. T. Weber. Towards mechanized program verification with separation logic. In J. Marcinkowski and A. Tarlecki, editors, *Computer Science Logic – 18th International Workshop, CSL 2004, 13th Annual Conference of the EACSL, Karpacz, Poland, September 2004, Proceedings*, volume 3210 of *Lecture Notes in Computer Science*, pages 250–264. Springer, September 2004.

AI4FM
A new project seeking challenges!

Gudmund Grov[1] and Cliff B Jones[2]

[1] School of Informatics, University of Edinburgh, UK,
ggrov@inf.ed.ac.uk
[2] School of Computing Science, Newcastle University, UK,
cliff.jones@ncl.ac.uk

Abstract. The "proof obligations" generated from many formal methods tend to be simple and can often be discharged by modern automatic theorem provers or SMT systems. However, those proof tasks that need hand –or interactive– intervention present a barrier to the use of formal methods. Theorem proving was one of the earliest challenges addressed by researchers in the area of Artificial Intelligence and enormous progress has been made in the provision of general purpose heuristics. The approach in the recently started AI4FM project is different: we hope to devise a system that will learn from an expert user how they tackle one interactive proof and then apply the discovered high-level strategy to other related proof tasks. We are fortunate in having access to many such problems through the DEPLOY project but are aware of the dangers of devising an overly specific approach. This short paper appeals for challenge problems from other sources.

1 Introduction

We have *just* embarked on a four-year research project (AI4FM) that will use Artificial Intelligence (AI) to tackle a core issue for "Formal Methods"[3] and we are keen to receive challenge problems.

Achieving verified software has been a dream since the birth of computer science [Jon03] and the importance of this objective has become ever greater with the increasing size and complexity of software.[4]

The use of formal methods has been successful in safety-critical domains, such as railway and aviation; gradually they are becoming increasingly popular in other sectors (e.g. Microsoft use formal methods to verify device drivers). A recent paper by Woodcock *et al.* [WLBF09] provides an up-to-date analysis of a significant number of industrial applications of formal methods. As the use of formal methods has spread beyond small groups of experts out to far larger

[3] 'Formal methods' use mathematics to specify, develop and reason about software and systems.

[4] We use the term software although the discussion here is valid for any application of formal methods, i.e. to generic system modelling as well as both hardware and software.

groups of industrial engineers, the importance of the availability of various sorts of "support tools" has been recognised. Such support tools include parsers/type checkers, full-blown theorem provers and decision procedures.

Formal methods are applied both *post facto* and in VxC *verified by construction*. As a shorthand, the former are often referred to as "bottom up" and the latter as "top down". Both approaches have their place and our decision to focus our current efforts on VxC has more to do with the applications that interest us than in any value judgement.

Top-down methods such as VDM [Jon90], B[Abr96] or Event-B[Abr10] tend to subscribe to a "posit and prove" pattern in which a designer posits a step of development and then seeks to justify it. By focusing on a particular style of development, a support tool can be built that generates "proof obligations" (POs) whose discharge justifies the correctness of a development step. For Event-B, one such support system is known as the "Rodin Tools"[5]— the generated POs are putative lemmas that need proof. In a carefully structured Event-B development, the theorem provers in the Rodin Tools will discharge the majority of the POs automatically; with skill and experience, users can get the percentage discharged automatically into the nineties. These results are typical of other choices of methods/tools. The remaining POs need to be discharged by interactive proof and this can be both time-consuming and challenging for industrial users; this in turn leads to proof being a bottleneck in industrial deployment.

There are, in fact, two approaches to dealing with the POs that require user interaction:

1. Follow a *modelling strategy*: change the model/abstraction in order to simplify the proofs, thus increasing the proportion of POs that are discharged automatically. For example, in [BY08], extra refinement steps are introduced due to known limitations of the automatic theorem provers.
2. Follow a *proof strategy*: accept the challenging POs and define a strategy for discharging them.

Both approaches are valid and useful and they can be seen as complementary. For example, the numbers quoted below most likely apply after several iterations of massaging the original model. A proof strategy could still be applied after the modelling strategy has reduced the numbers of undischarged POs.

It is the proof approach that we will take first in our AI4FM project. Our principal aim is to increase the repertoire of techniques for the proof-strategy approach by learning from proof attempts made by humans. We should make it clear that we are thinking of higher-level strategies than those normally coded in say HOL or Isabelle. In fact our model of the proof process is more like the sort of interactive proofs created in [JJLM91]; we are also aware that the progress in SMT research will have an influence on our approach. For example, we may incorporate SMT solvers as new tactics.

It should be remembered that the POs arising from formal methods tend to have different properties from "pure" mathematics.

[5] See www.event-b.org

1. There are often large numbers of detailed POs. To illustrate, the Paris Metro Line 14 and the Roissy Airport shuttle system were both developed using B [Abr07]; the former generated 27,800 POs (around 2,250 interactive) while the latter generated 43,610 POs (around 1,150 interactive).
2. POs tend to be less deep.
3. They often exhibit a "similarity", in the sense that they can be grouped into "families" and the same (high-level) proof approach can be successfully applied to all members of the family.

2 The AI4FM approach

There are two reasons of why a PO might not be discharged automatically: the putative lemma could be wrong (thus pinpointing a mistaken design decision[6]) or a true lemma has not been proved by the theorem prover because it is "not smart enough".

We have some data from industrial use that suggests failing POs fall into a relatively small number of distinct classes in the sense that one new idea will be key to discharging many POs. It is tempting to search for ever better heuristics but we plan to follow a different path in AI4FM. In many cases where a (correct) PO is not discharged automatically, an expert can easily see how to complete a proof. By exploring the nature of the POs within formal methods we believe that a higher degree of automation can be achieved by relying on expert intervention to do one proof, with the expectation that this would enable the system to discharge other POs in the same family.

Specifically, we hope to build a tool that will learn enough from one proof attempt to improve the chances of proving "similar" results automatically. By "proof attempt" we include things like the steps explored by the user (not just the chain of steps in the final proof). Thus it is central to our goal that we find *high-level* strategies capable of cutting down the search space in proofs.

Our hypothesis is:

> *we believe that it is possible (to devise a high-level strategy language for proofs and) to extract strategies from successful hand proofs that will facilitate automatic proofs of related POs.*

To achieve our goal we plan to analyse exemplar proofs (including their starting PO) using many *dimensions*. For example, we might separate information about data structures and approaches to different patterns arising from POs. Thus one proof (attempt) might be seen to use "generalise induction hypothesis" (e.g. adding an argument to accumulate values) about, say, sequences; a future use might involve a more complicated tree data structure — but if it has an induction rule, the same strategy might work. We hope to pick out other dimensions, such as the domain of the application that gave rise to the PO (e.g. does it relate to trains or to railway tracks?).

[6] An AI approach to help with these circumstances is discussed in [IGB10]

Designing a strategy language capable of capturing such properties (in an abstract form) is crucial to the success of AI4FM — one indication of the feasibility of such an approach is the earlier work on "proof critics" and "rippling" [BBHI05] — some early thoughts on a strategy language are presented in [JGB10] — and some *simple* examples in [BGJ09].

A key question for the design of the strategy language is the level of abstraction that will be used. We see two extreme points:

- a rather concrete description of a proof strategy (close to the tactic level), would not require much proof search when re-applying the strategy on POs in the same family – however, the size of the family would be rather small;
- much more abstract descriptions of proof strategies would capture far broader families of POs – however, they would require more proof search.

Thus, to reduce proof search, whilst keeping the families large, a language enabling proof strategy descriptions at different levels of abstractions seems desirable. HiProofs [DPT06] is an example to describe tactic proof using many levels, and we plan to build on this idea.

We would also need to provide tool support to both extract strategies from an exemplar interactive proof – and to interpret a strategy to discharge POs in the same family as the exemplar proof. We plan to build, at least the interpreter, on top of the Isabelle proof assistant. The advantage of Isabelle, is that it contains a meta-logic where we can, to some degree, develop our system – independent of the underlying method and logic.

Our solutions will rely heavily on heuristics — we do not believe that there are algorithmic solutions to most of our problems. Thus, as the project name suggests, our techniques will be heavily influenced by artificial intelligence. Particular areas of artificial intelligence we hope can help are

- *planning* and *proof planning* to find proofs from strategies – e.g. as in rippling discussed above;
- *machine learning* in order to
  - extract strategies from exemplar proofs. For example, *Explanation Based Generalisation/Learning* has previously been used to generalise subproofs for reuse [MS98]. However, something more general is probably required for our purposes;
  - discovering related POs, or finding a particular strategy for a sub-proof of a PO. We will need to use a form of pattern recognition in order to achieve this – which explores the various dimensions as discussed above.
- from the exemplar proof we may find "dead ends" in the search space, and use *search techniques* to rule them out from the target search space – i.e. strategies at the level of the search space.

3 We need more challenge portfolios

We are fortunate that our access through the DEPLOY project[7] to industrial users of the Rodin support tools will facilitate the capture of many difficult POs.

[7] See www.deploy-project.eu

We have already begun to find out how easy it is to analyse them into families that succumb to similar ideas to get their proofs to go through. But we are aware that it is always dangerous to base research on too narrow a base. We should therefore like to elicit challenge problems from other projects.

We can see three levels of useful access.

- Simple: we would be interested to receive POs generated from formal models of non-trivial computer systems — if these are beyond the power of the automatic theorem provers and/or SMT systems at hand, they might be interesting challenges for us — we are aware that even transferring a single model is not a simple file because we will need any base (data type) theories and, potentially, information about the logic used
- Valuable: it would be even more useful if we could get access to families of related proofs — we would be quite happy to get part of set (from which we try to learn strategy) — and then subject to independent scrutiny the question of whether the strategies that we devise would help with the unseen proof tasks
- Optimal: if we could, in addition to the above, receive a proof history including "this is where our TP got stuck" we would gain more insight into what is needed

We would appreciate it if anyone considering sending us material contacted Gudmund Grov[8] since dialogue is more likely to make the process work. We fully understand that industrial users might wish to disguise details of their models by changing the names of functions and/or state components.

Acknowledgements We would like to thank all members of the AI4FM project, in particular Alan Bundy. This work is supported by EPSRC grant (EP/H024204/1 and EP/H024050/1): '*AI4FM: the use of AI to automate proof search in Formal Methods*'.

References

[Abr96] J.-R. Abrial. *The B-Book: Assigning programs to meanings*. Cambridge University Press, 1996.

[Abr07] J.-R. Abrial. Formal methods: Theory becoming practice. *Journal of Universal Computer Science*, 13(5):619–628, 2007.

[Abr10] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.

[BBHI05] A. Bundy, D. Basin, D. Hutter, and A. Ireland. *Rippling: Meta-level Guidance for Mathematical Reasoning*, volume 56 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2005.

[BGJ09] Alan Bundy, Gudmund Grov, and Cliff B. Jones. An outline of a proposed system that learns from experts how to discharge proof obligations automatically. In *Proceedings of Dagstuhl Seminar 09381: Refinement Based Methods for the Construction of Dependable Systems*, 2009.

[8] See the AI4FM web page www.ai4fm.org or email ggrov@inf.ed.ac.uk

[BY08] Michael Butler and Divakar Yadav. An Incremental Development of the Mondex System in Event-B. *Formal Aspect of Computing*, 20(1):61–77, 2008.

[DPT06] Ewen Denney, John Power, and Konstantinos Tourlas. Hiproofs: A hierarchical notion of proof tree. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 155:341–359, 2006.

[IGB10] Andrew Ireland, Gudmund Grov, and Michael Butler. Reasoned Modelling Critics: Turning Failed Proofs into Modelling Guidance. In *Proceedings of ABZ'10*, number 5977 in LNCS, pages 189–202. Springer-Verlag, 2010.

[JGB10] Cliff B. Jones, Gudmund Grov, and Alan Bundy. Some facets of a strategy language for proofs. In *5th Automated Formal Methods workshop (AFM'10)*, July 2010. Also available as Edinburgh University, School of Informatics technical report EDI-INF-RR-1377.

[JJLM91] C. B. Jones, K. D. Jones, P. A. Lindsay, and R. Moore. *mural: A Formal Development Support System*. Springer-Verlag, 1991.

[Jon90] C. B. Jones. *Systematic Software Development using VDM*. Prentice Hall International, second edition, 1990.

[Jon03] Cliff B. Jones. The early search for tractable ways of reasonning about programs. *IEEE, Annals of the History of Computing*, 25(2):26–49, 2003.

[MS98] Erica Melis and Axel Schairer. Similarities and Reuse of Proofs in Formal Software Verification. In Barry Smyth and Pádraig Cunningham, editors, *Proceedings of the 4th European Workshop on Advances in Case-Based Reasoning (EWCBR-98)*, volume 1488 of *LNAI*, pages 76–87, Berlin, September 23–25 1998. Springer.

[WLBF09] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald. Formal Methods: Practice and Experience. *ACM Computing Surveys*, 41(4), Oct 2009.

A Review of Verification Benchmark Solutions Using Dafny

Derek Bronish and Bruce W. Weide

The Ohio State University, Columbus OH 43210, USA
{bronish,weide}@cse.ohio-state.edu

Abstract. Proposed solutions to a collection of software verification "challenge problems" have been undertaken by a group using Dafny. The techniques employed to solve these problems present insights into the Dafny specification and verification process. Solutions to key problems including binary search of an array and proof of correctness of data representation are reviewed, with observations about language design and modularity, among other issues.

1 Introduction

In an effort to focus and unify the efforts of the software verification community, the authors of [1] (including the two of us) have proposed a number of incremental "benchmarks." These challenge problems present a wide assortment of potential impediments to automated verification in a manner that progressively increases in difficulty. The benchmarks also elaborate on the data that should be provided in each solution. This includes code, specifications, verification conditions, buggy programs that the system can identify as incorrect, and so on. Each properly documented solution should be, in essence, a self-contained software verification case study.

Dafny [2] is a programming language whose semantics is defined by translation into Boogie [3], thus allowing verification conditions to be generated and then proven using Z3 [4]. The language features Java-like reference semantics and a unique selection of primitives, including generic `set` and `seq` collection types. Dafny-based solutions to all eight of the benchmark problems have recently been claimed [5] and made publicly available [6]. We describe three of these benchmark solutions and discuss the approach used in them.

The benchmark problems, and the specific guidelines for how solutions should be structured, are intended to establish a detailed and rigorous basis on which to compare and contrast different approaches to software verification. We do not seek to position ourselves as *de facto* official reviewers of proposed benchmark solutions simply on the grounds that we suggested the benchmarks. Instead, we hope that this paper will serve as a model for future benchmark solution submissions and analyses, giving a general idea of how a review might be reported and encouraging others to participate in this conversation. Other fields of study

thrive on reviews and responses as effective means for disseminating ideas[1]; benchmarks and challenge problems provide a similar opportunity in software engineering.

For each of the reviewed benchmarks, we quote the original problem statement from [1], present the Dafny solution, and discuss its merits. We conclude with general remarks about the Dafny approach and the lessons learned in undertaking this analysis.

2 Benchmark 2: Binary Search in an Array

> **Problem Requirements:** Verify an operation that uses binary search to find a given entry in an array of entries that are in sorted order.

Dafny does not provide arrays as a primitive in the programming language. Therefore, Benchmark 2 is where we first see Dafny's approach to user-defined data types. The benchmark does not require that the data type itself be verified, only that client code using it can be verified relative to the correctness of the data type. Issues of data representation are covered in later benchmarks; the issue for now is simply the *specification* of the array.

The Dafny `Array` component is shown in Figure 1. Note that for simplicity, the Dafny solution fixes the contents of an array to be `int`s. This is permissible; generics are not explicitly addressed until the fourth benchmark problem.[2] An array of integers is treated as a `seq` of `int`. This is indicative of a surprising characteristic of Dafny data types: it does not seem possible to specify the behavior of an array without making a commitment to its representation. A `seq` is not an abstract mathematical entity that can be used to describe the behavior of `Array` operations; instead it is a real programmatic object that can only be used in specifications if it is an actual field of a particular `Array` representation.[3]

Given this state of affairs, one may wonder how a client could possibly write specifications (e.g., loop invariants) that involve `Array`s in a modular manner, without committing to any particular `Array` implementation. Dafny does not provide any special new mechanism for this, but rather suggests as a good practice that classes provide a sufficiently robust collection of "pure" functions to facilitate client specification writing.

`Get` is an example of one of these "mathematical" functions, whose bodies are expressions rather than statements in the programming language. Although there is currently no compiler for Dafny, these functions are intended to be executable. One interesting question is how exactly these will be implemented by a Dafny compiler. One may also wonder what performance characteristics `Array` will offer when represented as a `seq`.

---

[1] One famous example from cognitive science is Chomsky's review of Skinner [7].

[2] Actually, Benchmark 3 calls for a generic datatype, but the Dafny solution does not meet this requirement.

[3] Dafny does allow "ghost" variables, but this amounts to the same problem for purposes of present discussion; a ghost variable is treated identically to a non-ghost variable (i.e., via explicit updates).

```
class Array {
  var contents: seq<int>;
  method Init(n: int);
    requires 0 <= n;
    modifies this;
    ensures |contents| == n;
  function Length(): int
    reads this;
  { |contents| }
  function Get(i: int): int
    requires 0 <= i && i < |contents|;
    reads this;
  { contents[i] }
  method Set(i: int, x: int);
    requires 0 <= i && i < |contents|;
    modifies this;
    ensures |contents| == |old(contents)|;
    ensures contents[..i] == old(contents[..i]);
    ensures contents[i] == x;
    ensures contents[i+1..] == old(contents[i+1..]);
}
```

Fig. 1. Dafny `Array` component.

The binary search implementation is shown in Figure 2. We see that the loop can exit unnaturally (i.e., without the loop guard evaluating to false), and so the verification conditions must account for this. Unfortunately, the publicly-available Dafny solutions do not include verification conditions for inspection. Also of note is the arithmetic used to compute `mid`. It is carefully formulated to avoid overflow, which is a subtle bug that is specifically enunciated in the benchmark statement. It would have been nice to see this bug manifest in the code, and then be discovered by a failed verification, but built-in `int`s are unbounded in Dafny.

3 Benchmark 3: Sorting a Queue

> **Problem Requirements:** Specify a user-defined FIFO queue ADT that is generic (i.e., parameterized by the type of entries in a queue). Verify an operation that uses this component to sort the entries in a queue into some client-defined order.

In this benchmark, the issue of parameterizing the sorting operation by an ordering is deemed to be "central," and the invariants involved for even straightforward sorting algorithms are non-trivial.

In Dafny, one can define a function only by writing an explicit function body. This means it is not possible to specify a generic comparison function. So, the proposed solution proceeds by fixing the queue contents to be `int`s and

```
method BinarySearch(a: Array, key: int) returns (result: int)
  requires a != null;
  requires (forall i, j :: 0 <= i && i < j && j < a.Length() ==>
                a.Get(i) <= a.Get(j));
  ensures -1 <= result && result < a.Length();
  ensures 0 <= result ==> a.Get(result) == key;
  ensures result == -1 ==>
            (forall i :: 0 <= i && i < a.Length() ==> a.Get(i) != key);
{
  var low := 0;
  var high := a.Length();

  while (low < high)
    invariant 0 <= low && high <= a.Length();
    invariant (forall i :: 0 <= i && i < low ==> a.Get(i) < key);
    invariant (forall i :: high <= i && i < a.Length() ==>
                  key < a.Get(i));
    decreases high - low;
  {
    var mid := low + (high - low) / 2;
    var midVal := a.Get(mid);

    if (midVal < key) {
      low := mid + 1;
    } else if (key < midVal) {
      high := mid;
    } else {
      result := mid; // key found
      return;
    }
  }
  result := -1;  // key not present
}
```

Fig. 2. Binary search implemented in Dafny.

the ordering to be standard numeric "less than or equal to." This skirts the central issue of parameterization by an operation mentioned above, but leaves a challenging benchmark nonetheless.

Selection sort, implemented with a `RemoveMin` helper operation, is shown in Figures 3 and 4. Clearly, the solution is annotation-heavy. This is primarily due to the need to specify properties about permutations, e.g., that the outgoing (sorted) value of the queue is a permutation of the incoming value.

One can imagine that certain properties, e.g., two strings being permutations of each other, are so fundamental and useful that one may wish to isolate their mathematical definitions inside some sort of atomic unit that can be concisely referred to in specifications. This could be viewed as an analog of procedural abstraction, and indeed one could propose using Dafny's `function`s to serve in this role. However, it is difficult to see exactly where such a function should reside. `Queue` would be too limited a scope, because of course we may wish to use it with other user-defined data types that are implemented with a `seq` — for example a stack, or even the `Array` seen in Benchmark 2. A language system that uses pure mathematical modeling seems better suited to this kind of design, because it would allow such definitions to reside with the rest of the mathematical theory used in the modeling, rather than in some particular program component, where it is hard to reuse.

4 Benchmark 4: Layered Implementation of Map ADT

Problem Requirements: Verify an implementation of a generic map ADT, where the data representation is layered on other built-in types and/or ADTs.

The Dafny solution shown in Figure 5 uses two `seq`s to represent the map: one to hold the keys and one to hold the associated values, with the association being established by matching indices.

At this point, issues of modularity and abstraction are brought to center stage. Consider, for example, the `FindIndex` method. Without even fully understanding the contract, an intuitive gloss of the method's name indicates that implementation details have been exposed here. From an abstract point of view, a map has no notion of "index," and so `FindIndex` is in principle surprising.

One could respond that `FindIndex` is intended to be "private." Dafny currently has no enforced access restriction; otherwise the authors would surely use it here. However, even if there were such a mechanism, there are further violations of representation encapsulation in the `Map` class. One example is the ensures clause of `Find`, which exposes the fact that the class contains fields `keys` and `values` of type `seq`. Imagine that one might prefer to implement `Map` using two `Queue`s in the representation. To describe the exact same behavior, the specifications would have to change so that all mentions of `keys` and `values` instead refer to `keys.contents` and `values.contents`, respectively.[4]

[4] In fact the situation becomes worse when one considers that there may be multiple implementations of `Queue`, each exposing its own representation in different ways and with different field names.

```
  method RemoveMin(q: Queue<int>) returns (m: int, k:int)
    requires q != null && |q.contents| != 0;
    modifies q;
    ensures |old(q.contents)| == |q.contents| + 1;
    ensures  0 <= k && k < |old(q.contents)| && old(q.contents[k]) == m;
    ensures (forall i :: 0 <= i && i < |q.contents| ==> m <= q.contents[i]);
    ensures q.contents == old(q.contents)[k+1..] + old(q.contents)[..k];
  {
    var n := |q.contents|;
    k := 0;
    m := q.Head();
    var j := 0;

    while (j < n)
      invariant j <= n;
      invariant q.contents == old(q.contents)[j..] + old(q.contents)[..j];
      invariant 0 <= k && k < |old(q.contents)| && old(q.contents)[k] == m;
      invariant (forall i ::0<= i && i < j ==> m <= old(q.contents)[i]);
      decreases n-j;
    {
      call x:= q.Dequeue();
      call q.Enqueue(x);
      if ( x < m) {k := j; m := x;}
      j:= j+1;
    }

    j := 0;
    while (j < k)
      invariant j <= k;
      invariant q.contents == old(q.contents)[j..] + old(q.contents)[..j];
      decreases k-j;
    {
      call x := q.Dequeue();
      call q.Enqueue(x);
      j:= j +1;
    }

     call m:= q.Dequeue();
   }
}
```

Fig. 3. Dafny `RemoveMin` implementation.

```
method Sort(q: Queue<int>) returns (r: Queue<int>, perm: seq<int>)
  requires q != null;
  modifies q;
  ensures r != null && fresh(r) && |r.contents| == |old(q.contents)|;
  ensures (forall i, j :: 0 <= i && i < j && j < |r.contents| ==>
            r.Get(i) <= r.Get(j));
  ensures |perm| == |r.contents|; // ==|pperm|
  ensures (forall i: int :: 0 <= i && i < |perm| ==>
            0 <= perm[i] && perm[i] < |perm|);
  ensures (forall i, j: int :: 0 <= i && i < j && j < |perm| ==>
            perm[i] != perm[j]);
  ensures (forall i: int :: 0 <= i && i < |perm| ==>
            r.contents[i] == old(q.contents)[perm[i]]);
{ r := new Queue<int>;
  call r.Init();
  // initialize ghostvar p so p=<0,...,|q.contents|-1> (code omitted)
  perm:= []; ghost var pperm := p + perm;
  while (|q.contents| != 0)
    invariant |r.contents| == |old(q.contents)| - |q.contents|;
    invariant (forall i, j :: 0 <= i && i < j && j < |r.contents| ==>
                r.contents[i] <= r.contents[j]);
    invariant (forall i, j :: 0 <= i && i < |r.contents| && 0 <= j &&
                j < |q.contents| ==> r.contents[i] <= q.contents[j]);
    invariant pperm==p+perm&& |p|==|q.contents|&& |perm|==|r.contents|;
    invariant (forall i: int :: 0 <= i && i < |perm| ==>
                0 <= perm[i] && perm[i] < |pperm|);
    invariant (forall i:int::0<=i && i<|p|==> 0<=p[i] && p[i]<|pperm|);
    invariant (forall i, j: int :: 0 <= i && i < j && j < |pperm| ==>
                pperm[i] != pperm[j]);
    invariant (forall i: int :: 0 <= i && i < |perm| ==>
                r.contents[i] == old(q.contents)[perm[i]]);
    invariant (forall i: int :: 0 <= i && i < |p| ==>
                q.contents[i] == old(q.contents)[p[i]]);
    decreases |q.contents|;
  {
    call m,k := RemoveMin(q);
    perm := perm + [p[k]]; //adds index of min to perm
    p := p[k+1..]+ p[..k]; //remove index of min from p
    call r.Enqueue(m);
    pperm := pperm[k+1..|p|+1]+pperm[..k]+pperm[|p|+1..]+[pperm[k]];
  }
  //lemma needed to trigger axiom
  assert (forall i:int :: 0<=i && i < |perm| ==> perm[i] == pperm[i]);
}
```

Fig. 4. Dafny `Sort` implementation. A trivial portion of code that properly initializes the ghost `seq p` is omitted for presentation purposes, but is correctly annotated and verified in the full solution.

```
class Map<Key,Value> {
  var keys: seq<Key>;
  var values: seq<Value>;

  function Valid(): bool
    reads this;
  {
    |keys| == |values| &&
    (forall i, j :: 0 <= i && i < j && j < |keys| ==> keys[i] != keys[j])
  }
  method Find(key: Key) returns (present: bool, val: Value)
    requires Valid();
    ensures !present ==> key !in keys;
    ensures present ==> (exists i :: 0 <= i && i < |keys| &&
                                     keys[i] == key && values[i] == val);
  {
    call j := FindIndex(key);
    if (j == -1) {
      present := false;
    } else {
      present := true;
      val := values[j];
    }
  }
  method FindIndex(key: Key) returns (idx: int)
    requires Valid();
    ensures -1 <= idx && idx < |keys|;
    ensures idx == -1 ==> key !in keys;
    ensures 0 <= idx ==> keys[idx] == key;
  {
    var j := 0;
    while (j < |keys|)
      invariant j <= |keys|;
      invariant key !in keys[..j];
      decreases |keys| -j;
    {
      if (keys[j] == key) {
        idx := j;
        return;
      }
      j := j + 1;
    }
    idx := -1;
  }
}
```

Fig. 5. An implementation of a map component using two parallel `seq`s. Methods for initializing the map, adding values, and removing values have been omitted solely for conciseness of presentation.

This is symptomatic of the aforementioned coupling between abstract specifications and concrete realizations. In the comments accompanying their Benchmark 4 solution, the authors note that this problem would be easier to solve if "Dafny had a built-in map type that could be used in specifications," but of course one cannot expect such an entity to exist for every kind of data type one may need to implement. Pure mathematical modeling is a convenient way of avoiding these issues and decoupling code from specifications, but it introduces complexities — such as establishing a correspondence between the programmatic entities participating in the data representation and the model used to express the desired behavior [8, 9] — that Dafny is not yet engineered to handle.

5 Conclusion

The most important issue brought to light by these proposed benchmark solutions has to do with Dafny's treatment of mathematical modeling. Dafny components tightly couple specifications with their implementations — indeed there seems to be no mechanism for writing specifications in a separate code unit.

In Dafny, some "mathematical" entities such as sets and sequences are actually primitives in the programming language. They serve a dual role: as descriptors for component behavior, and as actual fields of the class implementing that behavior. For example, both `Array` and `Queue` are specified and represented with a Dafny `seq`. One advantage of this approach is that in situations where a particular property is needed both for specification and for implementation purposes, a single entity (a Dafny `function`) can be used to define it. One example of this is the `Get` function seen in Benchmark 2.

On the other hand, this conflation of code and mathematics also poses some threats to modularity. For example, consider the issue of multiple implementations. In Benchmark 3, we see that class fields (e.g., `contents`) are allowed to appear in client specifications. Now imagine that there were two implementations of `Queue`: one which has a `seq` called `contents`, and another which uses a `set` called `contents`, likely in conjunction with some other fields. Exchanging one implementation with the other would inadvertently change the meaning (or even the well-formedness) of client specifications. The recommended idiom is of course to decouple the implementation from the specifications by only mentioning functions rather than fields on the client side, but the proposed solution to Benchmark 3 provides evidence that this is not currently possible in all cases.[5]

As authors of [1], we are pleased that the verification community has begun to take on the challenges proposed therein. Dafny's ability to address all eight benchmarks is impressive and appreciated; unfortunately space does not permit a presentation or review of all solutions here.

The collection, cataloging, and analysis of multiple solutions to the verification benchmark problems provide a valuable opportunity for detailed analyses and comparison. This work presents an initial discussion; we hope others follow.

[5] The authors note in the comments of their solution to Benchmark 3 that attempting to use `Get` instead of referring to `contents` does not work.

6 Acknowledgements

First and foremost we thank K. Rustan M. Leino and Rosemary Monahan for their impressive work on the Dafny benchmark solutions and their willingness to share their results, insights, and answers to questions with us. The authors are also grateful for the constructive feedback from Bruce Adcock, Paolo Bucci, Harvey M. Friedman, Wayne Heym, Jason Kirschenbaum, Bill Ogden, Murali Sitaraman, Hampton Smith, Aditi Tagore, and Diego Zaccai. This material is based upon work supported by the National Science Foundation under Grants No. ECCS-0931669, DMS-0701260 and CCF-0811737. Any opinions, findings, conclusions, or recommendations expressed here are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

1. Weide, B.W., Sitaraman, M., Harton, H.K., Adcock, B.M., Bucci, P., Bronish, D., Heym, W.D., Kirschenbaum, J., Frazier, D.: Incremental benchmarks for software verification tools and techniques. In: Verified Software: Theories, Tools, and Experiments (VSTTE). (2008) 84–98
2. Leino, K.R.M.: Specification and verification of object-oriented software. In Broy, M., Sitou, W., Hoare, T., eds.: Engineering Methods and Tools for Software Safety and Security. Volume 22 of NATO Science for Peace and Security Series D: Information and Communication Security. IOS Press (2009) 231–266 Summer School Marktoberdorf 2008 lecture notes.
3. Barnett, M., Chang, B.Y.E., Deline, R., Jacobs, B., Leino, K.R.: Boogie: A modular reusable verifier for object-oriented programs. In: Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005, volume 4111 of Lecture Notes in Computer Science, Springer (2006) 364–387
4. Moura, L.D., Bjørner, N.: Z3: An efficient SMT solver. In: Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). (2008)
5. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR-16). (2010)
6. Microsoft Research: The Boogie Project Codeplex Page. In: http://boogie.codeplex.com/SourceControl/list/changesets. (last accessed March 29th 2010)
7. Chomsky, N.: A review of B.F. Skinner's Verbal Behavior. Language **35**(1) (1959) 26–58
8. Hoare, C.A.R.: Proof of correctness of data representations. Acta Informatica **1**(4) (1972) 271–281
9. Sitaraman, M., Weide, B.W., Ogden, W.F.: On the practical need for abstraction relations to verify abstract data type representations. IEEE Trans. Softw. Eng. **23**(3) (1997) 157–170

VACID-0: Verification of Ample Correctness of Invariants of Data-structures, Edition 0

K. Rustan M. Leino and Michał Moskal

Microsoft Research, Redmond, WA, USA
{leino,micmo}@microsoft.com

Manuscript KRML 209, 21 July 2010.

Abstract. This paper defines a suite of benchmark verification problems, to serve as an acid test for verification systems that reason about full functional correctness of programs with non-trivial data-structure invariants. Solutions to the benchmarks can be used to understand similarities and differences between verification tools and techniques. The paper also gives a procedure for scoring the solutions.

0 Introduction

There are many program verification systems today, and we expect many more to be developed in the next decade. The systems differ in what programming language they handle, in the underlying logic and the specification features that they support, in the style in which specifications are written, in how one interacts with the system, and in the level of automation provided by the system. Yet, in the end, the tools all have a similar goal: to increase our confidence level in particular algorithms or pieces of code, and to do so in a convincing way without spending too much effort. So, how do we measure and compare the capabilities of various tools, how do we measure the improvements of a given tool over time, and how we compare how well tools live up to the common goal?

In this paper, we propose a set of challenging verification benchmarks, aimed to facilitate such measurements and comparisons. These benchmarks are for modular verification of functional correctness of programs where the data structures themselves rely on interesting consistency conditions. That is, the benchmarks were chosen as ones that involve useful and non-trivial data structures, not just non-trivial algorithms. Please note that these benchmarks are unlikely to be suitable for model-checking, testing, or extended static checking tools, with limited specification capabilities.

We propose the following five benchmarks, mostly freshman data structures.

0. **Constant-time spare array**, where `init`, `get` and `set` operations all take constant time. A simple warm-up problem exercising invariants over arrays.
1. **Composite pattern**, which has been proposed before. There is an update in the middle of a tree pointer structure, and then the algorithm walks the tree up to fix the invariant.
2. **Binary heap** is expected to be implemented with a integer-indexed data structure. We use heap-sort as a test-harness.

3. **Union-find** data structure for maintaining partitioning of a set. We use a maze generation algorithm (essentially Kruskal's spanning tree construction) as a test-harness.
4. **Red-black trees**. The classic. Likely most difficult in the set.

While challenging, we believe the benchmarks are solvable with state-of-the-art tools. We have started verification of some using the Dafny [6] and VCC [1] tools, and convinced ourselves that the others can be also verified. Thus, we expect the comparison to happen based on the performance and annotation overhead of the tools (see Sect. 1 below).

The benchmarks do not rely on object-orientation, type parameters, pointer arithmetic, unsafe features, particular concurrency handling, etc. It is thus possible to implement them in C or Java-like languages. The data structures and signatures that we give are imperative, thus performing verification in a functional programming language may be challenging.

The selection of benchmarks is clearly rather arbitrary. It would be ideal to have some objective benchmarks provided by outsiders, however we do not know of any such benchmarks. The situation is similar in the theorem prover competitions (CAST, SMT-COMP, SAT-race), where authors of the tools are allowed to submit benchmarks. The idea is, of course, that authors of *all* tools can submit benchmarks, giving a good mix in the end. For example, SMT-LIB welcomes any benchmarks, but during competition the benchmarks arising from actual applications (as opposed to random or hand-crafted) are given higher weights. In that spirit, a verification benchmark is valuable as long as it is something users will want to verify. We we believe that this is the case for all benchmarks presented here. The initial feedback we have received about our benchmarks from the verification community was overall very positive.

Each benchmark comes with a signature of operations available on the data structure and a "test harness"—a short program measuring the completeness of specifications of the data structure. We intentionally do not give exact specifications of the data structure operations, as these may vary between tools.

We also give a scoring scheme for the benchmarks. The scores are broken down so that it is possible to get points even if not all interesting properties to verify have been verified. Parts of the scores come from conciseness of specifications and other overhead for the proof, as well as from the performance of the realized verification tool.

Our goals are:

- to benchmark the ability of verification tools to verify the correctness of algorithms/-executable code that involve non-trivial data structure invariants, and
- to benchmark their support for modular checking and facilities for abstraction.

Having been inspired by a first suite of verification benchmarks, presented at VSTTE 2008 by Weide *et al.* [10], and having attempted to solve those using Dafny [7], we have tried to give more precise problem statements, in particular through the test harness mechanism. Also, unlike the VSTTE 2008 benchmarks, we do not aim at incrementality (*i.e.*, that one benchmark would build on another); rather, we give independent problems where the harness makes use of the abstract specification of the data structure in the same benchmark.

We expect the benchmarks to be suitable tasks to most of the tools listed below (the list is of course not expected to be exhaustive): the general purpose interactive provers (Coq, Isabelle/HOL, ACL2); interactive verification environments (Ynot, KeY, KIV, RESOLVE); symbolc execution tools (jStar, VeriFast, VeriCool); and deductive verifiers using automatic provers in the backend (Dafny, VCC, Chalice, Jahob, and the Why tools (including Krakatoa and Frama-C)). Three efforts to undertake the benchmarks are already underway, by Jan Smans (using VeriCool 3 [8]) and by us (using VCC and Dafny). We have setup an open-source project[0], to collect solutions.

1 Requirements and Scoring

We expect *modular* solutions, *i.e.*, one should verify correctness of a reasonably abstract specification (each problem specifies at least one such abstraction), and then verify the harness using only the specification, not the implementation of the data structure.

We also expect that the verification includes the safety of all operations, like absence of null dereferences, array bounds errors, and other precondition violations.

Each problem is scored on a scale from 0 to 100 points, inclusive. The description of the problem gives scores for various subtasks, which add up to 80. Usually some of the subtasks depend on others. If any points are scored for any of those subtasks, then additional points are added:

- 5 points are added if the solution guarantees termination, *i.e.*, total correctness of the test harness and the other code is proven
- $6 - 2 \ln \frac{g}{p}$ or 10 points (whichever is less) are added, where g is the number of tokens in the specification and all the additional lemmas/guidance needed for the verifier to verify the problem in batch mode and p is the number of tokens in the executable part of the program (including the harness)
- $5 - \frac{1}{2} \ln t$ or 5 points (whichever is less) are added, where t is the number of seconds the verifier needs in batch mode to verify the problem (including the harness)

Should the score for a benchmark come out negative (*i.e.*, the benchmark takes a couple of million years to verify or requires millions of lines of annotations), then the score is to be treated as 0 for that benchmark.

The result of the benchmark suite is the average number of points scored in individual benchmarks, where any benchmark not attempted counts as 0.

The definition of token will necessarily vary between languages. For languages with C-like syntax, we provide a script[1] to count them (which may need to be tweaked for a particular verifier).

The runtime should be measured as the wall-clock time on a modern (as of 2010) machine, *i.e.*, about a 3GHz PC. To reward parallel processing, any number of processors/cores are allowed, provided they all run concurrently during the interval wall-clock time reported. The measurement must, however, be done on an actual machine—it should not be constructed as a number that one, in principle, could obtain.

[0] `http://vacid.codeplex.com/`

[1] `http://research.microsoft.com/~moskal/count-tokens.zip`

Ideally, we would like to measure the tool's response time to the user while the user is developing the specification. That would mean computing the response time for a single method (or whatever the unit of specification/verification is) for a *failed* verification. However, since this is difficult to compare between different tools, we have instead settled for measuring the time needed for a successful verification of each entire benchmark.

Let us shortly motivate the scoring. As for the subtasks, we think that solving the problem at all is more important than annotation overhead or performance (this is similar to various theorem prover competitions). In particular, solving another subtask should not lower the overall score. This is why each subtask is worth at least 20 points. Other than that the amounts of points are just a guess, in some cases educated by our own attempts to solve these, or similar, problems.

As for the overhead, let us look at some data points. The extended static checking tools (*e.g.*, ESC/Java [4]) have an overhead of about 0.1 (which gives 10 points). Tools aiming at functional verification are unlikely to do any better. Automatic deductive tools, like Dafny and VCC, tend to have overhead roughly between 1 and 4 (6 to 4 points). Finally, the impressive L4.verified [5] project, using interactive Isabelle/HOL prover to verify functional correctness of an operating system, is reporting overhead of 20 (which gives 0 points). Any higher overhead gives negative points on our simple benchmarks.

As for the time, for interactive work in an IDE the response time should be in the sub-second range, and thus the 1 second overall time gives the maximal 5 points. Otherwise, the verification tool is likely to be used similarly to the compiler, which gives warnings about the code in the ballpark of 1 to 10 minutes (3 to 2 points). Finally, the response time of over an hour seems to effectively prevent development of specification, and thus the overall time between 1 and 6 hours gets between 1 and 0 points. Anything longer gives negative points.

The pseudo-code that we use below to describe data structures, the signatures of operations, and the various test harnesses is similar to Java or C#. We use type **`uint`** to denote the unsigned integers used to index into arrays, mostly to clearly distinguish them from values stored in the arrays (*i.e.*, **`int`**'s); however, there is no requirement on solutions to use unsigned types. It is permitted and encouraged (but currently does not yield any extra points) to use a generic type instead of **`int`**, and a particular value instead of `0` for all collection data-structures. We use a statement **`assert`**`(P)` to say that we require condition `P` to be verified to always hold (at least for full-points solutions; see exact description with each task).

1.0 Solution Description

To facilitate comparison between tools, solutions should report for each benchmark:

- which verification tasks were solved
- whether termination was proven
- the size in tokens of executable code and annotations; include separate numbers for the data-structure itself and the harness (the assertions in harness count as executable code, as they only emulate a use of the data-structure)

- the time it takes to verify the solution with multiple cores, and also with a single core
- if applicable, the longest verification time of a method (also multi- and single-core)
- test machine configuration (in particular, number of cores, if they were used, as well as any unusual resource requirement)
- the approximate time and number of people it took to develop the annotations, together with some indication of familiarity of the specification developers with the verification tools and techniques
- and, of course, the score computed as indicated above.

2 Constant-Time Sparse Array

For this benchmark, the task is to implement and verify an array where all three basic operations (create, get, and set) take constant time (Exercise 2.12 in [0]; see also [9] p. 271). Any memory requested from the underlying memory allocator (typically `malloc()` or **new**) should be treated as containing arbitrary values. The solution should use three arrays: one for the actual values stored in the array, and two more for marking which indices are already initialized, as in the program below.

```
class SparseArray {
  int val[MAXLEN];
  uint idx[MAXLEN], back[MAXLEN];
  uint n;
  static SparseArray create(uint sz) {
    SparseArray t = new SparseArray();
    n = 0;
    return t;
  }
  int get(uint i) {
    if (idx[i] < n && back[idx[i]] == i) return val[i];
    else return 0;
  }
  void set(uint i, int v) {
    val[i] = v;
    if (!(idx[i] < n && back[idx[i]] == i)) {
      assert(n < MAXLEN); // (*), see Verification Tasks
      idx[i] = n; back[n] = i; n = n + 1;
    }
  }
}
```

Verify that the program above meets the usual abstract interface of an array. The `get()` and `set()` methods should require the index to be within bounds, and the `create()` method may require that `sz <= MAXLEN`.

In a language like Java, the internal arrays allocated by the class above would already be 0-initialized; however, our benchmark stipulates that the implementation is not

allowed to make use of that fact in the verification. One way to ensure that is to make the constructor take the arrays as input.

The following test harness should be verified without looking at the particular implementation of the array.

```
void sparseArrayTestHarness() {
  SparseArray a = create(10), b = create(20);
  assert(a.get(5) == 0 && b.get(7) == 0);
  a.set(5, 1); b.set(7, 2);
  assert(a.get(5) == 1 && b.get(7) == 2);
  assert(a.get(0) == 0 && b.get(0) == 0);
}
```

2.0 Verification Tasks

0. Verify the correctness of the array implementation against your specification, assuming `n < MAXLEN` at the place marked with `(*)` in the code. Also, verify the correctness of the test harness using the specifications of the array. *50 points*.
1. As above, but without the assumption `(*)`. In other words, verify that the assertion holds (which ensures that the program does not reference the array outside its bounds). *30 points*.

3 Composite Pattern

This benchmark, which has been used as a specification and verification challenge at SAVCBS 2008 [3], involves a set of nodes connected acyclicly via `parent` links. In addition to graph structure information, each node stores an integer `val` as well as the sum of the `val` fields in the tree rooted at the node. The key is to keep these `sum` fields up-to-date.

A client is allowed access to any node in the set. That is, clients can hold pointers to any node. When the value of a node is updated, all relevant `sum` fields must be updated as well. This can be accomplished using recursion or by a loop like:

```
void update(int v) {
  int diff = v - val;
  val = v;
  for (CompositeNode p = this; p != null; p = p.parent) {
    p.sum = p.sum + diff;
  }
}
```

The signature of the class is:

```
class CompositeNode {
  CompositeNode parent;
  CompositeNode left, right;
```

```
  int val, sum;
  static CompositeNode create(int v);
  void addChild(CompositeNode child);  // connect 'this' as the parent of 'child'
  void dislodge();                     // disconnect 'this' from its parent
  void update(int v);
}
```

Here, we have limited nodes to two children, but programs are also allowed to support an unbounded number of children. The precondition of `addChild()` should prevent cycles from being created and prevent a node from getting more children than the implementation supports. Method `dislodge()` severs a node's tie to its parent. All methods must keep all `sum` fields up-to-date.

```
void compositeHarness()
{
  CompositeNode a = create(5), b = create(7);
  a.addChild(b);
  assert(a.sum == 12);
  b.update(17);
  assert(a.sum == 22);

  CompositeNode c = create(10);
  b.addChild(c); b.dislodge();
  assert(b.sum == 27);
}
```

It is allowed to modify the specification above to introduce an "manager" object to keep track of disjoint trees of composites.

3.0 Verification Tasks

0. Verify correctness of the harness. *80 points*.

4 Binary Heap

A binary min-heap (see [2], Chapter 6) is a nearly full binary tree, where the nodes maintain the *heap property*, that is, each node is smaller than each of its children. The heap should be stored in an integer-indexed collection (*e.g.*, an array). The following three operations should be provided:

```
class Heap {
  static Heap create(uint sz);
  void insert(int e);
  int extractMin();
}
```

The `create(sz)` method creates a new heap of maximum capacity `sz` (this restriction is optional). The `insert()` method should allow inserting an element multiple times so that `extractMin()` will return it multiple times.

The test harness consists of two procedures to be verified separately: a simple implementation of heap sort, and one use case.

```
void heapSort(int[] arr, uint len) {
  uint i;
  Heap h = create(len);
  for (i = 0; i < len; ++i) h.insert(arr[i]);
  for (i = 0; i < len; ++i) arr[i] = h.extractMin();
}
void heapSortTestHarness() {
  int[] arr = { 42, 13, 42 };
  heapSort(arr, 3);
  assert(arr[0] <= arr[1] && arr[1] <= arr[2]);
  assert(arr[0] == 13 && arr[1] == 42 && arr[2] == 42);
}
```

One may find the operation that "bubbles-up" an entry upon insertion to be similar in spirit to the composite pattern: we break the invariant at some point, and then move up fixing it. The motivation for including both benchmarks is that the composite pattern is supposed to be implemented with pointer structures, and the heap with an array (or another integer-indexed collection). Applications are likely to need both and thus we test for both.

4.0 Verification Tasks

0. Verify the that the heap sort returns an array that is sorted (in particular, verify the first assertion in the harness). *40 points*.
1. Verify that the heap represents a multiset, and thus that the heap sort produces a permutation of the input (in particular, verify the second assertion). *40 points*.

5 Union-Find

This benchmark makes use of the union-find data structure:

```
class UnionFind {
  static UnionFind create(uint sz);
  uint getNumClasses();
  int find(int a);
  void union(int a, int b);
}
```

Method `create()` creates a union-find data structure consisting of `sz` elements, identified by the integers from `0` to less than `sz`. Initially, each element is in an equivalence class by itself; that is, it is its own representative. In other words, the number of

equivalence classes, which is returned by `getNumClasses()`, is initially `sz`. As usual, `find()` returns the representative element for a given element, and `union()` merges two equivalence classes.

The test harness constructs a random maze. The maze has size `n` times `n`. Thus, it is a graph with `n*n` nodes. The maze will be a spanning tree of that graph. The maze construction makes use of union-find as illustrated here:

```
uint rand();
void buildMaze(uint n) {
  UnionFind u = create(n*n);
  while (u.getNumClasses() > 1) {
    uint x = rand() % n, y = rand() % n, d = rand() % 2;
    uint w = x, z = y;
    if (d == 0) w++; else z++;
    if (w < n && z < n) {
      int a = y*n + x, b = w*n + z;
      if (u.find(a) != u.find(b)) {
        output edge ((x,y), (w,z));
        u.union(a, b);
      }
    }
  }
}
```

The verification should be performed for an arbitrary `rand()` function, even if an implementation is actually provided with the solution. The test harness shown above is acceptable for verifying partial correctness, but needs to be suitably modified in order to stand a chance of scoring points for termination.

The maze is considered correct if it is a spanning tree. More precisely, the task in this benchmark is to show that all `n*n` nodes are reachable (either from a particular node or that nodes are pairwise connected) and that there are `n*n-1` edges. (From these two properties, it follows that nodes are uniquely reachable.)

5.0 Verification Tasks

0. Verify the correctness of the maze creation. *60 points*.
1. Include path compression and node balancing in the implementation of the union-find algorithms. *20 points*.

The task 1 involves the two standard optimizations that make union-find efficient. We did not split them further, as node balancing is expected to be very easy to implement and verify. For task 0, any correct implementation of union-find is acceptable.

6 Red-black Trees

A red-black tree [2] is a commonly used kind of binary search tree where each node, in addition to the usual data and pointers, carries a bit of information referred to as the

color (traditionally, either *red* or *black*). Tree operations maintain approximate balance by using rotation guided by colors of nodes. The task is to implement and verify a red-black tree providing a dictionary interface, like the one below:

```
class RedBlackTree {
  static RedBlackTree create(int defaultValue);
  void replace(int key, int value);
  void remove(int key);
  int lookup(int key);
}
```

The method `create(d)` creates a new dictionary mapping all keys to `d`. `replace(k, v)` replaces the current value associated with `k` by `v`. `remove(k)` is functionally equivalent to `replace(k, d)`, where `d` is the default value provided upon construction, but see Sect. 6.0 below. Finally, `lookup(k)` returns the current value associated with `k`.

The following test harness should be verified without looking at the particular implementation of red-black trees; that is, the interface above should be specified using appropriate abstraction, *e.g.*, a map or a set of pairs.

```
void redBlackTestHarness() {
  RedBlackTree a = create(0), b = create(1);
  a.replace(1, 1); b.replace(1, 10);
  a.replace(2, 2); b.replace(2, 20);
  assert(a.lookup(1) == 1 && a.lookup(42) == 0);
  assert(b.lookup(1) == 10 && b.lookup(42) == 1);
  a.remove(1); b.remove(2);
  assert(a.lookup(1) == 0 && a.lookup(42) == 0);
  assert(b.lookup(2) == 1 && b.lookup(42) == 1);
}
```

6.0 Verification Tasks

0. Prove the correctness of the test harness and tree implementation against the specifications of the tree (it is allowed to implement `remove()` by a call to `replace()`). *40 points*
1. Implement `remove()` so it really removes a node from the tree ([2], Section 13.4). *20 points*.
2. Prove the red-black balancing invariant, that is, that a red node cannot be a parent of another red node and that every path from a leaf to the root contains the same number of black nodes. *20 points*.

Like in the heap, the color fix-up operation is similar in spirit to the composite pattern. However here, the invariant involved is much more complex—the benchmark is designed to test scalability of the tool.

7 Conclusion and Future Work

The VACID-0 test provides a basis for comparing verification tools on implementations of non-trivial data structures. We hope that these and other solutions will lend themselves to interesting comparisons, and that they will help shape future editions of verification benchmarks.

References

0. Alfred V. Aho and John E. Hopcroft. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1974.
1. Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In *Theorem Proving in Higher Order Logics (TPHOLs 2009)*, volume 5674 of *LNCS*, pages 23–42. Springer, 2009. Invited paper.
2. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2nd revised edition edition, September 2001.
3. Robby (editor). Proceedings of the Seventh International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2008). Technical Report CS-TR-08-07, University of Central Florida, 2008.
4. Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 234–245. ACM, May 2002.
5. Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *POPL*, pages 207–220. ACM, 2009.
6. K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR-16*, LNCS. Springer, 2010. To appear.
7. K. Rustan M. Leino and Rosemary Monahan. Dafny meets the verification benchmarks challenge. In *VSTTE 2010*, LNCS. Springer, 2010. To appear.
8. Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In *ECOOP*, volume 5653 of *LNCS*, pages 148–172. Springer, 2009.
9. James A. Storer. *An Introduction to Data Structures and Algorithms*. Birkhauser Boston, 2001.
10. Bruce W. Weide, Murali Sitaraman, Heather K. Harton, Bruce Adcock, Paolo Bucci, Derek Bronish, Wayne D. Heym, Jason Kirschenbaum, and David Frazier. Incremental benchmarks for software verification tools and techniques. In Natarajan Shankar and Jim Woodcock, editors, *Verified Software: Theories, Tools, Experiments, Second International Conference, VSTTE 2008*, volume 5295 of *LNCS*, pages 84–98. Springer, October 2008.