Electronic Communications of the EASST
Volume 23 (2009)

Proceedings of the
Ninth International Workshop on
Automated Verification of Critical Systems
(AVOCS 2009)

A decidable class of verification conditions for programs with higher order store

Nathaniel Charlton Bernhard Reus

17 pages

Guest Editor: Markus Roggenbach
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

A decidable class of verification conditions for programs with higher order store

Nathaniel Charlton **Bernhard Reus**

n.a.charlton@sussex.ac.uk bernhard@sussex.ac.uk
School of Informatics
University of Sussex

Abstract: Recent years have seen a surge in techniques and tools for automatic and semi-automatic static checking of imperative heap-manipulating programs. At the heart of such tools are algorithms for automatic logical reasoning, using heap description formalisms such as separation logic. In this paper we work towards extending these static checking techniques to languages with procedures as first class citizens. To do this, we first identify a class of entailment problems which arise naturally as verification conditions during the static checking of higher order heap-manipulating programs. We then present a decision procedure for this class and prove its correctness. Entailments in our class combine *simple symbolic heaps*, which are descriptions of the heap using a subset of separation logic, with (limited use of) *nested Hoare triples* to specify properties of higher order procedures.

Keywords: higher order store, verification, nested triples, separation logic

1 Introduction

Program verification systems can be characterised according to the range and depth of properties they prove about programs, and the degree of user assistance they require. Some systems are designed for proving "full functional correctness", that is, for proving that a program performs exactly the function it is intended to. Proving full functional correctness involves a significant annotation burden, and difficult verification conditions must be discharged interactively by users due to the limitations of automated theorem proving. On the other hand systems such as ESC/Java [FLL$^+$02] check only a limited range of properties (e.g., memory safety), which enables them to run automatically, using decision procedures or automated theorem provers. Such systems have been termed *lightweight*, and behave much like enhanced type checkers.

Among the systems for lightweight verification are several successful tools which use *separation logic* [Rey02], an extension of classical logic which facilitates local reasoning about heap-manipulating programs. These tools include Smallfoot [BCO05a] and its descendants and HIP [NDQC07]. Separation logic provides two main ingredients: the *separating conjunction* $\star$ and the *frame rule*. The formula $P \star Q$ describes heaps which can be split into two disjoint parts, one part satisfying P and the other satisfying Q. Other new subformulae include *emp*, which describes the empty heap, and the *points-to* formulae such as $x \mapsto \{\!|f_1 : y, f_2 : \_ |\!\}$ which describes a heap containing exactly one object, pointed to by variable x; this object has at least the field f_1, with value y, and the field f_2 which may have any value. The frame rule states that if a command

C satisfies the triple $\{P\}C\{Q\}$, and does not modify any variables free in R, then it also satisfies $\{P \star R\}C\{Q \star R\}$. This embodies the idea that parts of the heap which the command does not access (here described by R) remain unchanged.

We work towards extending these lightweight program checking tools to higher order languages, and in particular those with higher order store. A programming language is called *higher order* if it treats procedures as first class values, so that procedures can be returned by expressions, passed as parameters etc. and then invoked later when needed. Additionally, such a language is said to have *higher order store* if procedures can be stored in updateable variables. Higher order store presents a serious obstacle to verification: since the procedure referred to by a particular name (e.g. stored in a particular variable) can change as the program executes, it is no longer adequate to use a single static specification for each procedure. *Nested triples* (e.g. [SBRY09, HYB05]) are a natural idea for reasoning about such procedure updates. In this approach, specifications (Hoare triples) for procedures can appear as ordinary assertions about program state, and hence triples can be *nested inside other triples*, in the pre- or post-conditions.

Our contribution in this paper is to give a decision procedure for a class of verification conditions (VCs) which supports the combined use of the two logical features we have described, separation logic and nested triples. By providing these features, our class includes VCs which arise naturally during the verification of heap-manipulating imperative programs with higher order store. We prove the correctness of our decision procedure, and illustrate by example the role it plays within the static program checker we are developing. The class of VCs we decide is strongly restricted; however the result we give is, to our knowledge, the first decidability result for verification conditions involving nested triples.

1.1 Related work

We have already mentioned lightweight tools which check properties of programs specified using separation logic. Decision procedures for fragments of separation logic are known, and used in such tools; [BIP08] gives arguably the most general of these. None of the existing tools allow the use of nested triples, however. Our aim is to use our decision procedure and its future extensions to extend such tools to languages with higher order store.

Various works build verification systems on top of powerful theorem proving environments such as Isabelle and Coq, which can be used for proving full functional correctness. [MN05] uses higher order logic to reason about the heap, and formalises (imperative) programs and proofs about them in Isabelle. [ZKR08] also uses higher order logic and Isabelle. We believe that these two works could be extended to address higher order store. [NMB08] shows using Coq that separation logic reasoning principles are admissible in Hoare Type Theory, an extension of a dependently and polymorphically typed functional language (allowing higher order functions) with Hoare-style specifications for state. We expect that our results, or adaptions thereof, can also be usefully integrated with such systems: decision procedures can be used to automatically discharge many proof obligations, leaving only the most involved proofs to be done interactively.

Several papers have investigated the theoretical aspects of adapting separation logic for use with higher order programs. [SBRY09, BRSY08, RS06, HYB05] study programs with higher order store and use logics with nested triples, while [BTY06, BY07, KAB$^+$09] study higher order procedures in the style of Idealised Algol and use a form of *specification logic*. This existing

```
procedure MAIN $GetEvent\ UserProcTemplate$
    new $ctr$;
    $ctr.cnt := 0$;
    $SendToAll :=$ INCR $ctr$;
    new $eventBuffer$;
    call $GetEvent\ eventBuffer$;
    $t := eventBuffer.eventType$;

    while $t \neq$ END do                                ▷ This is the main event-handling loop
        if $t$ = TALK then                               ▷ A connected user has typed a message
            $m := eventBuffer.message$;
            call $SendToAll\ m$
        else if $t$ = CONNECT then                       ▷ A new user has joined the chatroom
            $u := eventBuffer.newUserInfo$;
            $NewUserProc := UserProcTemplate\ u$;
            $OldSend := SendToAll$;
            $SendToAll :=$ APPLY2 $OldSend\ NewUserProc$
        call $GetEvent\ eventBuffer$;
        $t := eventBuffer.eventType$

    dispose $eventBuffer$;
    dispose $ctr$

procedure INCR $x\ y$
    $n := x.cnt$;  $x.cnt := n+1$

procedure APPLY2 $F\ G\ a$
    call $F\ a$;  call $G\ a$
```

Figure 1: The (idealised) outline of a chatroom server, programmed using higher order store.

research does not discuss the issue of decidability i.e. the issue of which kinds of verification conditions can be proved or falsified fully automatically.

2 An example verification problem with higher order store

Fig. 1 gives an imperative program which uses higher order store to implement (the idealised outline of) a server for a chatroom. In addition to three fixed procedures INCR, APPLY2 and MAIN, the program uses dynamically created procedures as follows. Each connected user is encapsulated in a procedure which, when invoked on a message argument m, sends m to the appropriate user. The server maintains a procedure *SendToAll* which is run whenever a connected user types a message (a 'Talk' event). *SendToAll* broadcasts its message argument to all connected users, by invoking the procedure encapsulating each user, and also updates a counter which records the number of messages sent. When a new user joins the chatroom (a 'Connect' event) the program *updates the procedure SendToAll*, replacing it with one which additionally broadcasts to the new user. The new *SendToAll* is built from the old one, the procedure encapsulating the new user, and the higher order procedure APPLY2. Using procedures like APPLY2 in a Curried style provides a useful way to dynamically generate new code; as in functional languages we write (partial) application of (Curried) procedures simply as juxtaposition.

Using nested triples and separation logic we can reason naturally about the example program.

Specifications for fixed procedures

$$\forall x \forall y\ \{x \mapsto \{\!|\mathit{cnt} : \_|\!\}\}\ \textsc{incr}\ x\ y\ \{x \mapsto \{\!|\mathit{cnt} : \_|\!\}\}$$

$$\forall F \forall G \forall a\ \Big\{\forall x \{\Psi\}\ F\ x\ \{\Psi\}\ \wedge\ \forall x \{\Psi\}\ G\ x\ \{\Psi\}\ \wedge\ \Psi\Big\}\ \textsc{apply2}\ F\ G\ a\ \{\Psi\}$$

$$\forall\ \mathit{GetEvent}\ \forall\ \mathit{UserProcTemplate}\ \{\Phi\}\ \textsc{main}\ \mathit{GetEvent}\ \mathit{UserProcTemplate}\ \{\underline{\mathrm{D}}\}$$

Loop invariant for loop in $\textsc{main}$ (Φ^{INV})

$$\begin{aligned}
& \forall x\ \{\underline{\mathrm{D}} \star \mathit{ctr} \mapsto \{\!|\mathit{cnt} : \_|\!\}\}\ \mathit{SendToAll}\ x\ \{\underline{\mathrm{D}} \star \mathit{ctr} \mapsto \{\!|\mathit{cnt} : \_|\!\}\} \\
\wedge\ & \forall a \forall x\ \{\underline{\mathrm{D}}\}\ \mathit{UserProcTemplate}\ a\ x\ \{\underline{\mathrm{D}}\} \\
\wedge\ & \forall \mathit{buf}\ \{\underline{\mathrm{D}} \star \mathit{buf} \mapsto \{\!||\!\}\}\ \mathit{GetEvent}\ \mathit{buf}\ \{\underline{\mathrm{D}} \star \mathit{buf} \mapsto \{\!|\mathit{eventType} : \_, X|\!\}\} \\
\wedge\ & \underline{\mathrm{D}} \star \mathit{eventBuffer} \mapsto \{\!|\mathit{eventType} : \_, X|\!\} \star \mathit{ctr} \mapsto \{\!|\mathit{cnt} : \_|\!\}
\end{aligned}$$

State after initialisation section of $\textsc{main}$ (Φ^{INIT})

$$\begin{aligned}
& \forall x\ \{\mathit{ctr} \mapsto \{\!|\mathit{cnt} : \_|\!\}\}\ \mathit{SendToAll}\ x\ \{\mathit{ctr} \mapsto \{\!|\mathit{cnt} : \_|\!\}\} \\
\wedge\ & \forall a \forall x\ \{\underline{\mathrm{D}}\}\ \mathit{UserProcTemplate}\ a\ x\ \{\underline{\mathrm{D}}\} \\
\wedge\ & \forall \mathit{buf}\ \{\underline{\mathrm{D}} \star \mathit{buf} \mapsto \{\!||\!\}\}\ \mathit{GetEvent}\ \mathit{buf}\ \{\underline{\mathrm{D}} \star \mathit{buf} \mapsto \{\!|\mathit{eventType} : \_, X|\!\}\} \\
\wedge\ & \underline{\mathrm{D}} \star \mathit{eventBuffer} \mapsto \{\!|\mathit{eventType} : t, X|\!\} \star \mathit{ctr} \mapsto \{\!|\mathit{cnt} : \_|\!\}
\end{aligned}$$

where X is the two entries “$\mathit{message} : \_,\ \mathit{newUserInfo} : \_$”, Ψ is $\underline{\mathrm{D}} \star \mathit{ctr} \mapsto \{\!|\mathit{cnt} : \_|\!\}$, and Φ is

$$\begin{aligned}
\underline{\mathrm{D}}\ \ \wedge\ & \forall \mathit{buf}\ \{\underline{\mathrm{D}} \star \mathit{buf} \mapsto \{\!||\!\}\}\ \mathit{GetEvent}\ \mathit{buf}\ \{\underline{\mathrm{D}} \star \mathit{buf} \mapsto \{\!|\mathit{eventType} : \_, X|\!\}\} \\
\wedge\ & \forall u\, \forall \mathit{msg}\ \{\underline{\mathrm{D}}\}\ \mathit{UserProcTemplate}\ u\ \mathit{msg}\ \{\underline{\mathrm{D}}\}
\end{aligned}$$

Figure 2: Memory safety specifications for the fixed procedures in Fig. 1.

For instance, the initialisation statement “$\mathit{SendToAll} := \textsc{incr}\ \mathit{ctr}$” in Fig. 1 satisfies the triple

$$\left\{\begin{array}{c} \forall x \forall y \{x \mapsto \{\!|\mathit{cnt} : \_|\!\}\} \\ \textsc{incr}\ x\ y \\ \{x \mapsto \{\!|\mathit{cnt} : \_|\!\}\} \end{array}\right\}\ \begin{array}{c} \mathit{SendToAll} := \\ \textsc{incr}\ \mathit{ctr} \end{array}\ \left\{\begin{array}{ccc} \forall x \forall y \{x \mapsto \{\!|\mathit{cnt} : \_|\!\}\} & & \forall y \{\mathit{ctr} \mapsto \{\!|\mathit{cnt} : \_|\!\}\} \\ \textsc{incr}\ x\ y & \wedge & \mathit{SendToAll}\ y \\ \{x \mapsto \{\!|\mathit{cnt} : \_|\!\}\} & & \{\mathit{ctr} \mapsto \{\!|\mathit{cnt} : \_|\!\}\} \end{array}\right\}$$

Here the program variable *ctr* appears free in the triple for *SendToAll*, nested in the postcondition: the value of *ctr* has been “hard-wired” into *SendToAll* via partial application of the Curried procedure $\textsc{incr}$. As explained earlier, the specification for $\textsc{incr}$ is lightweight: specifically, the pre- and post-conditions say which objects should be in the heap at which addresses, and which fields they should contain, but not what the values of the fields should be. We remark that with higher order programs one can do almost no reasoning without using universal quantifiers, as one needs to specify how procedures behave for *all* possible invocations.

Suppose we annotate our program with specifications for the three fixed procedures, plus an invariant for the loop in $\textsc{main}$, as in Fig. 2. We use the *assertion variable* $\underline{\mathrm{D}}$ to stand for whatever concrete data structure is used to store the necessary information about the connected users. Our verification of the code in Fig. 1 will not need any assumptions about this data structure, and so will be generic with respect to it; when one verifies implementations of *GetEvent* and *UserProcTemplate* (which we shall not do here), however, one will instantiate $\underline{\mathrm{D}}$ concretely.

We want our specifications checked by machine. A standard technique is to cut up the code into "straight-line" pieces, and from these and the annotations (specifications and loop invariant) generate verification conditions in the form of logical entailments. The VC generation step is not the subject of this paper: instead, we focus on the problem of how to automatically check the VCs once they have been generated. We just mention that we use forward reasoning rules akin to those of [BCO05b], rather than (backwards) weakest precondition rules.

The critical point is that for programs with higher order store the VCs contain (nested) triples. For instance (as can be shown by forward reasoning rules), after the initialisation section of MAIN the program state satisfies the formula Φ^{INIT} in Fig. 2. Thus one of the VCs is $\Phi^{\mathrm{INIT}} \models \Phi^{\mathrm{INV}}$, which corresponds to checking that the loop invariant is correctly established. Both Φ^{INIT} and Φ^{INV} contain triples for *SendToAll*. For this particular entailment, the main task is to show

$$\begin{array}{rl} & \forall x\, \{ctr \mapsto \{\!|cnt : \_|\!\}\}\ \mathit{SendToAll}\ x\ \{ctr \mapsto \{\!|cnt : \_|\!\}\} \\ \models & \forall x\, \{\underline{\mathrm{D}} \star ctr \mapsto \{\!|cnt : \_|\!\}\}\ \mathit{SendToAll}\ x\ \{\underline{\mathrm{D}} \star ctr \mapsto \{\!|cnt : \_|\!\}\} \end{array}$$

which can be done using a frame rule to add $\underline{\mathrm{D}}$ on the left of $\models$. Similarly, one must prove VCs to show that the loop body preserves the invariant, e.g. one must prove that the new procedure stored in *SendToAll* during (one branch of) the loop body still meets the appropriate specification, which amounts to checking the following entailment (where Ψ is $\underline{\mathrm{D}} \star ctr \mapsto \{\!|cnt : \_|\!\}$).

$$\begin{array}{rl} & \forall x\, \{\Psi\}\ \mathit{OldSend}\ x\ \{\Psi\} \quad \wedge \quad \forall x\, \{\underline{\mathrm{D}}\}\ \mathit{NewUserProc}\ x\ \{\underline{\mathrm{D}}\} \\ \wedge & \forall a \left\{ \begin{array}{l} \quad \forall x\, \{\Psi\}\ \mathit{OldSend}\ x\ \{\Psi\} \\ \wedge \quad \forall x\, \{\Psi\}\ \mathit{NewUserProc}\ x\ \{\Psi\} \ \wedge\ \Psi \end{array} \right\} \mathit{SendToAll}\ a\ \{\Psi\} \\ \models & \forall x\, \{\Psi\}\ \mathit{SendToAll}\ x\ \{\Psi\} \end{array} \tag{1}$$

Intuitively the triple for *SendToAll* in the antecedent says that *SendToAll* works as required *provided* the procedures *NewUserProc* and *OldSend*, from which it was built, also behave properly. The decidable class of VCs that we identify and solve contains (1) and similar entailments.

3 Formal foundations

In this section, we formalise a simple logic for reasoning about imperative programs with higher order store, which includes nested triples and separation logic features. Our VCs will then be entailment problems between formulae of this logic. In order to give formal semantics to the logic, however, we first give an overview of the higher order programming language we work with (we lack the space to give formal definitions for the language).

3.1 Overview of our programming language with higher order store

The language we work with is very similar to that used in Fig. 1: it is a `while` language with heap manipulation statements, extended with higher order storable procedures with value parameters, and is interpreted within the solution to the following system of domain equations.

$$\begin{array}{lcllcl} \mathsf{Heap} & = & \mathsf{Rec}_{\mathbb{Z}_{>0}}(\mathsf{Rec}_{\mathsf{FieldN}}(\mathbb{Z})) & \mathsf{Stack} & = & (\mathsf{Var} \to \mathbb{Z}) \times (\mathsf{PVar} \to \mathsf{Proc}) \\ \mathsf{State} & = & \mathsf{Stack} \times \mathsf{Heap} & \mathsf{Cmd} & = & \mathsf{State} \rightharpoonup (\mathsf{State} + \{\mathsf{fault}\}) \\ \mathsf{Proc} & = & \mathsf{VarList} \times \mathsf{Cmd} & & & \end{array}$$

Here $\mathsf{Rec}_S(\mathsf{A})$ is the set of records with fields from set S and values from domain A. Such a record is written $\{\!|f_1 = v_1,\ldots,f_n = v_n|\!\}$ (where $f_1,\ldots,f_n \in S$ and $v_1,\ldots,v_n \in \mathsf{A}$). We overload the $\star$ symbol to mean the "union" of disjoint records: $\{\!|f_1 = v_1,\ldots,f_n = v_n|\!\} \star \{\!|f'_1 = v'_1,\ldots,f'_m = v'_m|\!\}$ is the record $\{\!|f_1 = v_1,\ldots,f_n = v_n, f'_1 = v'_1,\ldots,f'_m = v'_m|\!\}$ if $\{f_1,\ldots,f_n\} \cap \{f'_1,\ldots,f'_m\} = \emptyset$ and is undefined otherwise. We write $R[f := v]$ for the record R updated (if $f \in dom(R)$) or extended (if $f \notin dom(R)$) with value v at field f (and overload this notation for updating functions too). $\mathsf{VarList}$ is the set of finite lists of variables from Var or PVar, where $x : xs$ is a list with head x and tail xs, and $[]$ is the empty list.

There are two kinds of values in our language: integers and procedures. We also distinguish two kinds of program variables, integer variables in Var (written in lower case), and procedure variables in PVar (written in upper case). Program states take the form (s,t,h) where s is the integer part of the stack (or simply *integer stack*), mapping Vars to integer values, t is the procedure stack, mapping PVars to procedures, and h is the heap. Heap cells in our language are similar to objects in e.g. JavaScript, in that they are essentially associative arrays, associating integer values to field names. Thus, procedures can be stored on the stack but not on the heap.

Observe that the domain equations above are recursive to accommodate higher order store: commands map states to states but states themselves contain commands, stored in procedure variables. Commands can also produce the special value fault which indicates a low-level error such as accessing or disposing a non-allocated heap address. Commands are deterministic; in particular the language's memory allocator is deterministic. Procedures comprise a list of variables (the parameters) and a command (the body); when invoked, procedures run in a new stack with default values for all variables.

By $[\![C]\!]$ we denote the interpretation of a program statement C into Cmd. The interpretation of an expression E in stack (s,t), which gives a value in $\mathbb{Z} + \mathsf{Proc} + \{\mathsf{fault}\}$, is written $[\![E]\!]^{\mathscr{E}}_{(s,t)}$. As usual when separation logic is used, there is no expression for accessing the heap; rather, values needed from the heap must first be read into variables using the statement form $v := E.f$ which reads field f at address E. The expression $E_P\ E_A$ denotes the (partial) application of a (Curried) procedure E_P to an argument expression E_A. This provides a useful way to dynamically generate new procedures. For example, a statement "$G := F\ a$" copies the procedure stored in F into G, additionally fixing the first parameter to the *current* value of a. Future changes to the variable a will have no effect on the procedure stored in G, so programs cannot use the stack to dynamically create new recursions as in Landin's knot. The fixed procedures of a program, however, can be (mutually) recursive.

Procedures can only be invoked if they are not expecting further parameters; otherwise fault results. Inherent in our treatment of procedures is that all parameters behave as value parameters, so that from the caller's point of view procedure calls do not modify any stack variables. Integer values can be returned via the heap but procedures cannot.

3.2 A simple logic with nested triples and separating conjunction

Fig. 3 gives the syntax and semantics of our logic. Let AVar be the set of assertion variables (introduced on page 4) such as $\underline{\mathrm{D}}$; these are interpreted by an environment $\rho : \mathsf{AVar} \to \mathscr{P}(\mathsf{Heap})$. Assertions are then interpreted w.r.t. a state (s,t,h) and such a ρ. When no assertion variables are present we omit ρ. The following definition gives our total correctness, fault-avoiding in-

$$\Phi \ ::= \ E \mapsto \{\!|N^*|\!\} \mid emp \mid \underline{\mathrm{I}} \mid \Phi \star \Phi \mid \Phi \wedge \Phi \mid \forall v\, \Phi \mid \{\Phi\}\, E\, \{\Phi\} \qquad N \ ::= \ f : E \mid f : \_$$

$$\llbracket \Phi_1 \wedge \Phi_2 \rrbracket_\rho = \llbracket \Phi_1 \rrbracket_\rho \cap \llbracket \Phi_2 \rrbracket_\rho \qquad \llbracket emp \rrbracket = \{(s,t,h) \mid h = \{\!||\!\}\} \qquad \llbracket \underline{\mathrm{I}} \rrbracket_\rho = \{(s,t,h) \mid h \in \rho(\underline{\mathrm{I}})\}$$

$$\llbracket \Phi_1 \star \Phi_2 \rrbracket_\rho = \{(s,t,h) \mid \exists h_1, h_2 \text{ s.t. } h = h_1 \star h_2, (s,t,h_1) \in \llbracket \Phi_1 \rrbracket_\rho \text{ and } (s,t,h_2) \in \llbracket \Phi_2 \rrbracket_\rho\}$$

$$\llbracket \forall v\, \Phi \rrbracket_\rho = \{(s,t,h) \mid \text{for all } n \in \mathbb{Z},\ (s[v := n],t,h) \in \llbracket \Phi \rrbracket_\rho\}$$

$$\llbracket E \mapsto \{\!|N^1,\ \ldots,\ N^k|\!\} \rrbracket = \left\{ (s,t,h) \;\middle|\; \begin{array}{l} dom(h) = \{\llbracket E \rrbracket^{\mathscr{E}}_{(s,t)}\} \text{ and for each } N_i = f : X \\ f \in dom(h(\llbracket E \rrbracket^{\mathscr{E}}_{(s,t)})) \text{ and if } X \text{ is an expression} \\ \text{(i.e. not } \_\text{) then } (h(\llbracket E \rrbracket^{\mathscr{E}}_{(s,t)}))(f) = \llbracket X \rrbracket^{\mathscr{E}}_{(s,t)} \end{array} \right\}$$

For $\llbracket \{\Phi_1\}E\{\Phi_2\} \rrbracket_\rho$ see Definition 1. E means any expression of the programming language.

Figure 3: Syntax and semantics for a simple logic with separating conjunction and nested triples.

terpretation of triples. This definition is non-standard in that it also requires the procedure E to behave "locally": *any* extra piece of heap h_I must be left untouched by the procedure.

Definition 1 Semantics of nested triples. We define $\llbracket \{\Phi_1\}E\{\Phi_2\} \rrbracket_\rho := S \times$ Heap where $S \subseteq$ Stack is all those stacks (s,t) such that:

1. $\llbracket E \rrbracket^{\mathscr{E}}_{(s,t)}$ has the form $([],c)$ where $c \in$ Cmd, and

2. for all disjoint heaps $h, h_I \in$ Heap, if $(s,t,h) \in \llbracket \Phi_1 \rrbracket_\rho$ then there exist a heap g and a stack (s_L,t_L) such that $c(s^0,t^0,h \star h_I) = (s_L,t_L,g \star h_I)$ where $(s,t,g) \in \llbracket \Phi_2 \rrbracket_\rho$.

Here the stack (s^0,t^0) maps all Vars to the default value 0, and maps all PVars to $([],d)$, where $d \in$ Cmd always faults. The callee's local stack (s_L,t_L) is thrown away when the procedure returns and so, as stated earlier, procedure calls do not modify any of the caller's variables. □

The interpretation of the other connectives is standard. Because procedure calls do not change the value of any variables, we can quantify *program* variables over nested triples, and do not need a separate set of auxiliary variables. For a quantified formula $\Phi = \forall x_1 \ldots \forall x_n \Psi$ we will write $\Phi(E_1,\ldots,E_n)$ for the instantiation $\Psi[x_1,\ldots,x_n \backslash E_1,\ldots,E_n]$, where $E_1,\ldots,E_n$ are any expressions.

Definition 2 Entailment between assertions. Let Φ_1, Φ_2 be assertions. We say that Φ_1 *entails* Φ_2, and write $\Phi_1 \models \Phi_2$, if: for all ρ : AVar $\to \mathscr{P}$(Heap), $\llbracket \Phi_1 \rrbracket_\rho \subseteq \llbracket \Phi_2 \rrbracket_\rho$. □

We list the properties of entailment between triples that we will need in our proofs.

Frame property. $\{P\}E\{Q\} \models \{P \star R\}E\{Q \star R\}$. Because our inner triples concern procedure calls, and we use only value parameters, the usual side condition concerning variable modification [Rey02] is not needed. In our setting the frame property follows directly from the semantics of Hoare triples: the heap h_I in Definition 1 accounts for the added invariant R. This approached is called "baking in" the frame property, and is borrowed from [BY07].

AVar-substitution property. If $T_1 \models T_2$ then $T_1[\underline{\mathrm{I}}\backslash\Phi] \models T_2[\underline{\mathrm{I}}\backslash\Phi]$. This is a typical substitution property. (Again no side condition concerning variable modification is needed.)

Consequence property. If $P_2 \models P_1$ and $Q_1 \models Q_2$ then $\{P_1\}E\{Q_1\} \models \{P_2\}E\{Q_2\}$. This holds in the same way that the consequence rule for basic Hoare logic holds.

Crossing Out property. If Θ does not depend on the heap and does not contain $x_1,\ldots,x_n$ free, then $\Theta \wedge \forall x_1 \ldots x_n\{\Theta \wedge P\}E\{Q\} \models \forall x_1 \ldots x_n\{P\}E\{Q\}$. This follows straightforwardly from the axiom (e5) in [HYB05] which also holds in our setting. This axiom states that $\Phi \rightarrow \{P\}E\{Q\}$ is equivalent to $\{\Phi \wedge P\}E\{Q\}$ provided Φ does not depend on the heap.

3.3 Simple symbolic heaps (SSHs)

We next define a simple class of formulae that we will work with, inspired by the *symbolic heaps* of [BCO05b], and set up some machinery for manipulating them.

Definition 3 Simple symbolic heaps. A *simple symbolic heap* (SSH) is a formula

$$\underline{\mathrm{I}}_1 \star \cdots \star \underline{\mathrm{I}}_k \star \bigstar_{i=1}^{n} v_i \mapsto \{\!|f_i^1 : v_i^1,\ \ldots,\ f_i^{m_i} : v_i^{m_i}|\!\}$$

where $\underline{\mathrm{I}}_1,\ldots,\underline{\mathrm{I}}_k \in \mathsf{AVar}$ are distinct, $f_i^1,\ldots,f_i^{m_i} \in \mathsf{FieldN}$ are distinct for each i, $v_1,\ldots,v_n \in \mathsf{Var}$ are distinct and each v_i^j is either an integer variable or the special symbol $\_$. (Here we allow $k=0$ and/or $n=0$, leaving just *emp* in the appropriate part of the formula.) Each assertion variable or points-to formula occurring in the formula is called a *spatial conjunct*. □

Definition 4 "Ensures" relation. We define a relation *ensures*, between spatial conjuncts of the kind found in SSHs, as follows.

- $u \mapsto \{\!|f^1 : u^1,\ldots,f^N : u^N|\!\}$ ensures $v \mapsto \{\!|g^1 : v^1,\ldots,g^M : v^M|\!\}$ if $u = v$ and for each entry $g^j : v^j$ there is an entry $f^l : u^l$ with $f^l = g^j$ and such that v^j is either u^l or $\_$.
- A formula $\underline{\mathrm{I}} \in \mathsf{AVar}$ ensures itself.

We say that an SSH Φ *ensures* an SSH Ψ if Φ,Ψ have the same number of spatial conjuncts, and each spatial conjunct of Ψ is ensured by a spatial conjunct of Φ.

We write $P =_{\mathrm{SSH}} Q$ when SSHs P and Q are syntactically equal (modulo the order of the spatial conjuncts, and modulo the order of fields in the points-to subformulae). We lift the $\star$ connective to give an operator on SSHs in the obvious way; note that this lifted operator is partial (e.g. $A = x \mapsto \{\!|f : y|\!\}$ is an SSH, but $A \star A$ is not). Finally we define a partial subtraction operator: $A -_{\mathrm{SSH}} B$ is the SSH C (unique up to $=_{\mathrm{SSH}}$ when it exists) such that for some SSH B' we have $A =_{\mathrm{SSH}} B' \star C$ and B' ensures B. (Where convenient, we shall treat partial operations as though they have an option type as in ML, that is, return either $\mathsf{Some}\ x$ or None.) □

Lemma 1 Properties of SSHs.

1. *The (syntactic) relation "ensures" exactly captures the (semantic) entailment relation between SSHs, i.e.* $\Phi \models \Psi$ *iff* Φ *ensures* Ψ.

2. *If $P_2 -_{SSH} P_1 = R$ then $P_2 \models P_1 \star R$. (Implication in the other direction fails in general.)*

3. *If P is an SSH containing no assertion variables, and $s : \mathsf{Var} \overset{\cong}{\to} \mathbb{Z}_{>0}$ (by which we mean that the integer stack s is a bijection between* Var *and $\mathbb{Z}_{>0}$) then there exists a heap h such that for all t, $(s,t,h) \in [\![P]\!]$.* □

We will use stacks $s : \mathsf{Var} \overset{\cong}{\to} \mathbb{Z}_{>0}$ frequently because, as noted in 3. above, they free us from the concern that a formula such as $x \mapsto \{\!|f : \_|\!\} \star y \mapsto \{\!|f : \_|\!\}$ cannot be satisfied in any heap just because x and y contain the same address (recall that $\star$ partitions the heap into *disjoint* pieces).

3.4 SSH-triples and their properties

Next we define some classes of Hoare triples based on SSHs. These classes will be used to define our decidable class of verification conditions.

Definition 5 Suitable pairs of SSHs. A pair (P,Q) of SSHs is *suitable* if, writing P and Q as

$$P \;=\; \underline{\mathrm{I}}_1 \star \cdots \star \underline{\mathrm{I}}_k \;\star\; \bigstar_{i=1}^{n} u_i \mapsto X_i \qquad Q \;=\; \underline{\mathrm{J}}_1 \star \cdots \star \underline{\mathrm{J}}_l \;\star\; \bigstar_{j=1}^{m} v_j \mapsto Y_j$$

each v_j appears somewhere in $u_1,\ldots,u_n$, and $\underline{\mathrm{I}}_1 \star \cdots \star \underline{\mathrm{I}}_k$ is equal (modulo order) to $\underline{\mathrm{J}}_1 \star \cdots \star \underline{\mathrm{J}}_l$. (The role played by suitable SSH pairs will become clear later, when we discuss Theorem 1.) □

Definition 6 SSH-triples.

- By a *level 0 SSH-triple about procedure F* we mean a triple $\forall x_1 \ldots x_n \{P\}\ F\ vs\ \{Q\}$ where (P,Q) is a suitable pair of SSHs, $F \in \mathsf{PVar}$, $vs \in \mathsf{VarList}$ are integer variables and $x_1,\ldots,x_n \in \mathsf{Var}$ occur in vs and are distinct.

- By a *level 1 SSH-triple about F* we mean a triple $\forall x_1 \ldots x_n\ \{T_1 \wedge \cdots \wedge T_k\ \wedge\ P\}\ F\ vs\ \{Q\}$ in which (P,Q) is a suitable pair of SSHs, $F \in \mathsf{PVar}$, $vs \in \mathsf{VarList}$ are integer variables, $x_1,\ldots,x_n \in vs$ are distinct and $T_1,\ldots,T_k$ are level 0 SSH-triples about distinct procedures not including F, in which $x_1,\ldots,x_n$ do not occur free. □

At two points later we will reduce an entailment problem P_1 to a simpler entailment problem P_2. As part of proving such a reduction correct, we will take a counterexample to the entailment P_2 and construct from it a counterexample to P_1. To make this reasoning work smoothly, we will work only with counterexamples that have a particular form, as per the following definition.

Definition 7 Convenient counterexamples. Consider a non-entailment $T_1 \wedge \cdots \wedge T_k \not\models T$, where $T_1,\ldots,T_k,T$ are level 1 SSH-triples containing assertion variables $\underline{\mathrm{I}}_1,\ldots,\underline{\mathrm{I}}_n$. We say that the non-entailment has *convenient counterexamples* if, for any fresh variables $a_1,\ldots,a_n$, there exist formulae $\Phi_1,\ldots,\Phi_n$ and a state (s,t,h) such that: each Φ_i is *emp* or $a_i \mapsto \{\!|\,|\!\}$, $s : \mathsf{Var} \overset{\cong}{\to} \mathbb{Z}_{>0}$, $(s,t,h) \in [\![(T_1 \wedge \cdots \wedge T_k)[\underline{\mathrm{I}}_1,\ldots,\underline{\mathrm{I}}_n \backslash \Phi_1,\ldots,\Phi_n]]\!]$ and $(s,t,h) \notin [\![T[\underline{\mathrm{I}}_1,\ldots,\underline{\mathrm{I}}_n \backslash \Phi_1,\ldots,\Phi_n]]\!]$. □

Convenient counterexamples (which really are counterexamples due to the AVar-instantiation property) are convenient in two ways. Firstly they provide states where $s : \mathsf{Var} \overset{\cong}{\to} \mathbb{Z}_{>0}$ which is useful as explained above, and secondly they provide counterexamples that contain only points-to conjuncts, with assertion variables having been eliminated by substitution. In fact, because we can eliminate assertion variables in this way, we will ignore them in our sketch proofs.

Now we come to our first theorem. We show that for any level 1 SSH-triple T about a procedure F, given *any* values for the integer variables and *any* interpretation of $\underline{\mathrm{I}},\underline{\mathrm{J}},\ldots$, we can construct a program for F that meets the specification T.

Theorem 1 Existence of models for level 1 SSH-triples. *Let T be a level 1 SSH-triple about F. For any s,ρ there exists t such that $(s,t,h)\in[\![T]\!]_\rho$ for all h.*

Sketch proof. Let s,ρ be arbitrary; T has the form $\forall x_1\ldots x_n\ \{T_1\wedge\cdots\wedge T_k\ \wedge\ P\}\ F\ vs\ \{Q\}$. Let us write P and Q as in Definition 5. We build a program $C_1;\ldots;C_n$ where each C_i deals with the subformula $u_i\mapsto X_i$ in an appropriate way, to build a heap satisfying Q. If Q contains nothing of the form $u_i\mapsto X'$ then C_i is the statement **dispose** u_i which deallocates the cell at address u_i. Otherwise, some $u_i\mapsto X'$ is present in Q and C_i writes the appropriate values into the fields of the cell at u_i, e.g. if Q contains $u_i\mapsto\{\!|f_1:y,f_2:z|\!\}$ then C_i will be $u_i.f_1:=y;u_i.f_2:=z$. Putting $t(F):=(vs,[\![C_1;\ldots;C_n]\!])$ we get $(s,t,h)\in[\![\forall x_1\ldots x_n\{P\}\ F\ vs\ \{Q\}]\!]_\rho$ for all h, which suffices. □

We will depend crucially on this theorem later, because to show *non*-entailments $T_A\not\models T_B$ we generally begin with a model of T_A and modify it in some way, so that it remains a model of T_A but is not a model of T_B. Note that the restrictions in Definition 5 are carefully chosen to make this theorem work: for instance, the triple $\{a\mapsto\{\!|f:\_|\!\}\}\ F\ vs\ \{b\mapsto\{\!|f:\_|\!\}\}$, in which the pair of SSHs is not suitable, is *not* satisfiable with respect to every s, but only when $s(a)=s(b)$. Intuitively this is because if the program for F (which cannot change b) wishes to allocate new heap cells, it must use whatever locations the allocator chooses, and cannot *demand* to be allocated a particular address b. The suitability condition allows assertion variables $\underline{\mathrm{I}}\in\mathsf{AVar}$ to be used only as invariants, which means that the program constructed in the proof of Theorem 1 can simply leave alone the parts of the heap corresponding to assertion variables.

4 Three decidable entailment problems involving nested triples

In this, the main section of the paper, we give decision procedures for three classes of entailment problem involving triples, and prove their correctness. The classes are of increasing difficulty and each algorithm builds on the previous one.

4.1 Deciding entailments between quantifier-free level 0 SSH-triples

Fig. 4 gives a simple procedure DECIDE-ENT-QF-0 which, using the syntactic operators “ensures” and $-_{\mathrm{SSH}}$, decides entailment problems $\{P_A\}\ F\ vs\ \{Q_A\}\models\{P_B\}\ F\ vs\ \{Q_B\}$ between two quantifier-free level 0 SSH-triples. The following lemma establishes the procedure’s correctness in the case that it returns *true*. The proof is easy, using the Frame and Consequence properties.

Lemma 2 *Let $T_A=\{P_A\}\ F\ vs\ \{Q_A\}$ and $T_B=\{P_B\}\ F\ vs\ \{Q_B\}$ be level 0 SSH-triples such that $P_B-_{SSH}P_A=R$ and $Q_A\star R$ ensures Q_B. Then $T_A\models T_B$.*

```
procedure DECIDE-ENT-QF-0( $\{P_A\}\ F\ vs\ \{Q_A\}$, $\{P_B\}\ F\ vs\ \{Q_B\}$ )
    if $P_B -_{\mathrm{SSH}} P_A =$ Some $R$ then return $(R \star Q_A$ ensures $Q_B)$
    else return $false$

procedure DECIDE-ENT-0( $\forall x_1 \ldots x_n \{P_A\}\ F\ ts\ \{Q_A\}$, $T_B$ )
    let $\{P_B\}\ F\ vs\ \{Q_B\} =$ FRESHEN$(T_B)$ in
        if INSTANTIATE($\forall x_1 \ldots x_n \{P_A\}\ F\ ts\ \{Q_A\}$, $vs$) has the form Some $\{P\}\ F\ vs\ \{Q\}$ then
            return DECIDE-ENT-QF-0($\{P\}\ F\ vs\ \{Q\}$, $\{P_B\}\ F\ vs\ \{Q_B\}$)
        else return $false$

procedure DECIDE-ENT-1( $T_1, \ldots, T_k$, $T$ )
    let $\{P_2\}\ F\ vs\ \{Q_2\} =$ FRESHEN$(T)$ in
        if exists $i$ such that $T_i$ is about $F$ then
            if INSTANTIATE($T_i$, $vs$) has the form Some $\{T_1' \wedge \cdots \wedge T_r' \wedge P_1\}\ F\ vs\ \{Q_1\}$ then
                if forall $j$ DECIDE-ENT-1($T_1, \ldots, T_k$, $T_j'$) then
                    return DECIDE-ENT-QF-0($\{P_1\}\ F\ vs\ \{Q_1\}$, $\{P_2\}\ F\ vs\ \{Q_2\}$)
                else return $false$ else return $false$ else return $false$
```

Figure 4: Procedures for deciding three increasingly complex entailment problems involving SSH-triples. For explanation of FRESHEN and INSTANTIATE see main text (pages 12 and 13).

Proof. Using the Frame property to add R, T_A entails $\{P_A \star R\}\ F\ vs\ \{Q_A \star R\}$. This entails T_B by the Consequence axiom, provided we have $P_B \vDash P_A \star R$ and $Q_A \star R \vDash Q_B$. These two entailments follow from the hypotheses by Lemma 1, using parts 2. and 1. respectively. □

However, to justify our claim that DECIDE-ENT-QF-0 is a decision procedure, we must also show that when *false* is returned, the entailment genuinely does not hold; to do this, we construct a command for F that satisfies T_A but not T_B. It is in constructing such counterexample programs that the main work of the present paper lies. The following lemmas prove the required non-entailments and also show the existence of convenient counterexamples; these will be needed later when we use DECIDE-ENT-QF-0 as a subroutine when solving entailments with quantifiers.

Lemma 3 *Let $T_A = \{P_A\}\ F\ vs\ \{Q_A\}$ and $T_B = \{P_B\}\ F\ vs\ \{Q_B\}$ be level 0 SSH-triples such that $P_B -_{SSH} P_A$ does not exist. Then $T_A \nvDash T_B$ with convenient counterexamples.*

Sketch proof. $P_B -_{\mathrm{SSH}} P_A$ does not exist, so there is some $v \mapsto \{\!|f^1 : v^1, \ldots, f^m : v^m|\!\}$ in P_A that is not ensured by any spatial conjunct of P_B. By Theorem 1 there exists (s,t,h) such that $s : \mathsf{Var} \xrightarrow{\cong} \mathbb{Z}_{>0}$ and $(s,t,h) \in [\![T_A]\!]$ (and thus $[\![F\ vs]\!]^{\mathscr{E}}_{(s,t)} = ([], c)$ for some $c \in \mathsf{Cmd}$). We build a program C which tests for the presence of the missing heap cell at v. Specifically, we define C to be $\{\textbf{var } x; C_1; \ldots; C_m\}$ where each statement C_j is constructed as follows.

$$C_j \quad := \quad \begin{cases} x := n.f^j & \text{if } v^j \text{ is } \_ \\ x := n.f^j;\ \textbf{if } x = n^j \textbf{ then skip else dispose } 0 & \text{otherwise} \end{cases}$$

Here n is a literal integer constant with the value $s(v)$, which we can use because we work w.r.t. a fixed s. Similarly each n^j is the literal integer constant $s(v^j)$. Now we "prepend" our program C to c: put $\hat{t} := t[F := ([\,], c \circ [\![C]\!])]$. It suffices to show $(s,\hat{t},h) \in [\![T_A]\!]$ and $(s,\hat{t},h) \notin [\![T_B]\!]$. $(s,\hat{t},h) \in [\![T_A]\!]$ follows from $(s,t,h) \in [\![T_A]\!]$ because the precondition P_A makes sure that everything "tested" by C is in place; the call to F will behave as c. But $(s,\hat{t},h) \in [\![T_B]\!]$ fails because we can choose a heap h' which satisfies P_B but does not have the "tested" cell in place; C will fault on heap h'. □

Lemma 4 *Let $T_A = \{P_A\}\ F\ vs\ \{Q_A\}$ and $T_B = \{P_B\}\ F\ vs\ \{Q_B\}$ be level 0 SSH-triples such that $P_B -_{SSH} P_A = R$ and $R \star Q_A$ does not ensure Q_B. Then $T_A \not\models T_B$ with convenient counterexamples.*

Sketch proof. $R \star Q_A$ not ensuring Q_B can happen in several ways; we sketch the case where a whole heap cell from Q_B is missing from $R \star Q_A$, i.e. Q_B contains some $v \mapsto X$ but $R \star Q_A$ contains no $v \mapsto X'$. (Other cases include e.g. when only a field, rather than a whole cell, is missing.)

By Theorem 1 there exists (s,t,h) such that $s : \mathsf{Var} \xrightarrow{\cong} \mathbb{Z}_{>0}$ and $(s,t,h) \in [\![T_A]\!]$ (and thus $[\![F\ vs]\!]^{\mathscr{E}}_{(s,t)} = ([\,], c)$ for some $c \in \mathsf{Cmd}$). It suffices to show $(s,t,h) \notin [\![T_B]\!]$. Since $s : \mathsf{Var} \xrightarrow{\cong} \mathbb{Z}_{>0}$ there exists a heap h' such that $(s,t,h') \in [\![P_B]\!]$. From $P_2 -_{\mathrm{SSH}} P_1 = R$ and Lemma 1 we have $P_2 \models P_1 \star R$, so there exist disjoint h_1, h_2 such that $(s,t,h_1) \in [\![P_A]\!]$, $(s,t,h_2) \in [\![R]\!]$ and $h_1 \star h_2 = h'$. Now, $c(s^0,t^0,h') = c(s^0,t^0,h_1 \star h_2) = (s_L, t_L, h'' \star h_2)$ where $(s,t,h'') \in [\![Q_A]\!]$. Thus $(s,t,h'' \star h_2) \in [\![Q_A \star R]\!]$; since $Q_A \star R$ contains no $v \mapsto X'$ it follows that $s(v) \notin dom(h'' \star h_2)$. Suppose for a contradiction that $(s,t,h) \in [\![T_B]\!]$. Then, since $(s,t,h') \in [\![P_B]\!]$, we can use the triple T_B to see that $c(s^0,t^0,h') = (s_L, t_L, g)$ where $(s,t,g) \in [\![Q_B]\!]$. Thus $s(v) \in dom(g)$. But $g = h'' \star h_2$, so $s(v) \in dom(h'' \star h_2)$ and we have a contradiction. □

Corollary 1 **Correctness of the procedure** DECIDE-ENT-QF-0 *(Fig. 4). The procedure* DECIDE-ENT-QF-0 *decides $T_A \models T_B$ for quantifier-free level 0 SSH-triples T_A, T_B. Furthermore, when $T_A \not\models T_B$ there are convenient counterexamples.*

4.2 Deciding entailments between level 0 SSH-triples

We now extend our methods to decide entailments between pairs of level 0 SSH-triples. Universal quantifiers appearing on the right of $\models$ are unproblematic: one can use the standard technique of replacing the quantified variables with fresh free variables. We will assume that this is done by a procedure FRESHEN. This leaves an entailment problem $T_A \models T_B$ of the form

$$\forall x_1,\ldots,x_n \{P_A\}\ F\ ts\ \{Q_A\} \quad \models \quad \{P_B\}\ F\ vs\ \{Q_B\} \tag{2}$$

between level 0 SSH-triples. The universal quantifiers appearing on the left of $\models$ are more difficult. The standard way to use a universally quantified formula $\forall x \Phi$ in a proof is to instantiate it i.e. choose an expression E and then make use of $\Phi[x \backslash E]$. In general, proving an entailment $\forall x \Phi \models \Psi$ might need us to instantiate x with several different expressions, and even where one instantiation suffices, finding the right E can be difficult.

However, recall that our use of quantifiers is quite restricted: as per Definition 6, each quantified variable x must appear as a parameter in the procedure call $F\ ts$. Thus the natural approach is to choose instantiations, if any exist, which "unify" $F\ ts$ with $F\ vs$. In our restricted setting, this method turns out to suffice, as the following results show.

Lemma 5 *If there do not exist* $v_1,\ldots,v_n \in \mathsf{Var}$ *such that* $ts[x_1,\ldots,x_n \backslash v_1,\ldots,v_n] = vs$*, or if such exist but* $T_A(v_1,\ldots,v_n)$ *is not an SSH-triple, then* (2) *fails with convenient counterexamples.*

Sketch proof. We sketch very briefly the case where no such $v_1,\ldots,v_n$ exist. Theorem 1 provides an interpretation c for F which fulfils the antecedent of (2). Using this, we construct an interpretation c' which faults if the values of its parameters match vs, and otherwise behaves as c. Using c' in place of c clearly falsifies the consequent of (2), but crucially it also preserves the truth of the antecedent; the non-existence of $v_1,\ldots,v_n$ is exactly what is required to show this. □

Theorem 2 *Let* $v_1,\ldots,v_n \in \mathsf{Var}$ *be such that* $ts[x_1,\ldots,x_n \backslash v_1,\ldots,v_n] = vs$ *and* $T_A(v_1,\ldots,v_n)$ *is an SSH-triple. If* $T_A(v_1,\ldots,v_n) \vDash T_B$ *fails with convenient counterexamples then so does* $T_A \vDash T_B$.

Sketch proof. We assume $T_A(v_1,\ldots,v_n) \nvDash T_B$ with convenient counterexamples and prove $T_A \nvDash T_B$ with convenient counterexamples, by constructing a program for F that satisfies T_A but not T_B.

Because $T_A(v_1,\ldots,v_n) \nvDash T_B$ with convenient counterexamples, there exists (s,t,h) such that $s : \mathsf{Var} \xrightarrow{\cong} \mathbb{Z}_{>0}$ and $(s,t,h) \in [\![T_A(v_1,\ldots,v_n)]\!]$ (and thus $t(F) = ([p_1,\ldots,p_m],c)$ for some $c \in \mathsf{Cmd}$) but $(s,t,h) \notin [\![T_B]\!]$. By Theorem 1 there also exists t' such that $(s,t',h) \in [\![T_A]\!]$ where $t'(F) = ([p_1,\ldots,p_m],c')$ for some command $c' \in \mathsf{Cmd}$. Thus, we have two models: one for T_A, and another for $T_A(v_1,\ldots,v_n) \wedge \neg T_B$. Our idea is to merge these into a model for $T_A \wedge \neg T_B$. Here it is crucial that Theorem 1 works for *any* integer stack: if the two models had different integer stack components, the merging would not be possible. We define $\hat{t} := t[F := ([],d)]$ where the command d examines its parameters and decides whether to behave as c or as c'.

$$d(S,T,H) := \begin{cases} c(S,T,H) & \text{if } S(p_1),\ldots,S(p_m) = s(v_1),\ldots,s(v_m) \\ c'(S,T,H) & \text{otherwise} \end{cases}$$

It suffices to show $(s,\hat{t},h) \in [\![T_A]\!]$ but $(s,\hat{t},h) \notin [\![T_B]\!]$. Firstly we explain why $(s,\hat{t},h) \in [\![T_A]\!]$. If the parameters do not match $s(v_1),\ldots,s(v_m)$ then d behaves like c' and the result follows from $(s,t',h) \in [\![T_A]\!]$; if the parameters do match, one combines $p_1 = s(v_1),\ldots,p_m = s(v_m)$ and $(s,t,h) \in [\![T_A(v_1,\ldots,v_m)]\!]$ to obtain the required result. Secondly, $(s,\hat{t},h) \notin [\![T_B]\!]$ follows from $(s,t,h) \notin [\![T_B]\!]$ because if the parameters match $s(v_1),\ldots,s(v_m)$ then d behaves like c. □

The other direction, if $T_A(v_1,\ldots,v_n) \vDash T_B$ then $T_A \vDash T_B$, is obvious. Informally the preceding theorem shows that when an instantiation is discovered, it is *the* correct instantiation. Identifying the appropriate $v_1,\ldots,v_n$ is a straightforward unification problem, so let us assume that a procedure INSTANTIATE does this, returning either Some T where T is the triple T_A with the appropriate instantiation already performed, or None. Armed with INSTANTIATE we can now give, in Fig. 4, an algorithm DECIDE-ENT-0 which decides entailments between level 0 SSH-triples.

Corollary 2 **Correctness of the procedure** DECIDE-ENT-0 *(Fig. 4). The procedure* DECIDE-ENT-0 *decides* $T_A \vDash T_B$ *for level 0 SSH-triples* T_A, T_B*. Furthermore, when* $T_A \nvDash T_B$ *there are convenient counterexamples.*

We emphasise that in general, computing instantiations in this way is insufficient. Given triples $T_1 = \forall x\{true\}\ F\ x\ \{a \neq 1 \vee x = 2\}$ and $T_2 = \{true\}\ F\ b\ \{a \neq 1 \vee b \neq 2\}$ we have $T_1 \vDash T_2$, but the obvious instantiation $T_1(b)$ does not entail T_2. Non-intuitively one does have $T_1(a) \wedge T_1(b) \models T_2$, however. The existence of such cases is what makes Theorem 2 non-obvious and non-trivial.

On the other hand, the method used to prove Theorem 2 does extend to some more general cases. Consider for instance entailment problems of the form $\forall xy\{x \mapsto \{\!|f : y|\!\} \star P_A\}\ F\ x\ \{Q_A\} \models \{a \mapsto \{\!|f : b|\!\} \star P_B\}\ F\ a\ \{Q_B\}$. Here y (resp. b) does not appear as a parameter, but the procedure F can safely access y (resp. b) nevertheless, by looking into the heap at x (resp. a). Thus, we can construct a body for F that tests y against a constant value, and the technique used to prove Theorem 2 still works. Hence we can conclude that the only useful instantiation for y is b.

4.3 Deciding level 1 entailment problems

Finally we are in a position to address *level 1 entailment problems*, that is, entailments which have the form $T_1 \wedge \cdots \wedge T_k \models T$ where $T_1, \ldots, T_k$ are level 1 SSH-triples about distinct procedures and T is a level 0 SSH-triple. Suppose we wish to prove something of the following form:

$$\forall x\{R\}\ G\, x\ \{S\} \ \wedge\ \Big\{\forall x\{R'\}\ G\, x\ \{S'\} \ \wedge\ P\Big\}\ F\ y\ \{Q\} \quad \models \quad \{P'\}\ F\ y\ \{Q'\}$$

Here we need to prove that $F\ y$ has some desired behaviour (unconditionally), but the specification we have available for $F\ y$ (on the left of $\models$) only applies *provided* G satisfies a particular specification $\forall x\{R'\}\ G\ x\ \{S'\}$. Our idea is to split our argument into subproofs: 1. $\forall x\{R\}\ G\, x\ \{S\} \models \forall x\{R'\}\ G\, x\ \{S'\}$ and 2. $\{P\}\ F\ y\ \{Q\} \models \{P'\}\ F\ y\ \{Q'\}$. If these two subproofs can be done, then we can use the Crossing Out property to combine them into the required proof.

The algorithm DECIDE-ENT-1, given in Fig. 4, makes precise this idea. In general, part 1. of the proof attempt can require further splitting, which leads to a recursive algorithm. The following theorem shows that this approach is both sound when it succeeds, and complete, so that when the approach fails, a counterexample to the entailment exists.

Theorem 3 *Let $\forall x_1 \ldots x_n\ \{T_1' \wedge \cdots \wedge T_r' \wedge P_A\}\ F\ ts\ \{Q_A\}$ be a level 1 SSH-triple, let $T_1, \ldots, T_k$ be a list of level 1 SSH-triples for distinct procedures not including F and let $\{P_B\}\ F\ vs\ \{Q_B\}$ be a level 0 SSH-triple. Consider the following entailment problems.*

$$\text{(A)}\quad T_1 \wedge \cdots \wedge T_k \models T_1' \wedge \cdots \wedge T_r' \qquad\qquad \text{(B)}\quad \forall x_1 \ldots x_n\{P_A\}\ F\ ts\ \{Q_A\} \models \{P_B\}\ F\ vs\ \{Q_B\}$$

$$\text{(C)}\quad T_1 \wedge \cdots \wedge T_k \ \wedge\ \forall x_1 \ldots x_n\{T_1' \wedge \cdots \wedge T_r' \wedge P_A\}\ F\ ts\ \{Q_A\} \quad \models \quad \{P_B\}\ F\ vs\ \{Q_B\}$$

Then: 1. If (A) fails with convenient counterexamples, then so does (C). 2. If (A) holds and (B) fails with convenient counterexamples, then (C) fails with convenient counterexamples. 3. If (A) and (B) hold then (C) holds.

Sketch proof. For 1.: There exists a witness (s,t,h), where $s : \mathsf{Var} \overset{\cong}{\to} \mathbb{Z}_{>0}$, for the non-entailment $T_1 \wedge \cdots \wedge T_k \not\models T_1' \wedge \cdots \wedge T_r'$. We make use of (s,t,h) to construct a countermodel in which the antecedent of (C) is true but the consequent is false: specifically, our countermodel is $(s,\hat{t},h)$ where $\hat{t} = t[F := (ts, \llbracket \mathbf{dispose}\ 0 \rrbracket)]$.

To see that the consequent triple is false in $(s,\hat{t},h)$ observe that there exists a heap h' such that $(s,t,h') \in \llbracket P_B \rrbracket$ (this follows from $s : \mathsf{Var} \overset{\cong}{\to} \mathbb{Z}_{>0}$), but running F always causes a fault. Now we show that the antecedent is true in $(s,\hat{t},h)$. The triples $T_1 \wedge \cdots \wedge T_k$ are not “about” F, so their truth is not broken by the "update" we have made at F. It remains to show that $(s,\hat{t},h) \in [\![\forall x_1 \ldots x_n\{T'_1 \wedge \cdots \wedge T'_r \wedge P_A\}\ F\ \mathit{ts}\ \{Q_A\}]\!]$; this follows from three facts: $(s,t,h) \notin [\![T'_1 \wedge \cdots \wedge T'_r]\!]$, $T'_1,\ldots,T'_r$ are not about F, and $x_1,\ldots,x_n$ do not appear free in $T'_1,\ldots,T'_r$.

For 2.: There exists a witness (s,t,h) for $\forall x_1 \ldots x_n\{P_A\}\ F\ \mathit{ts}\ \{Q_A\} \not\models \{P_B\}\ F\ \mathit{vs}\ \{Q_B\}$ where $s : \mathsf{Var} \xrightarrow{\cong} \mathbb{Z}_{>0}$. Again we use (s,t,h) to construct a countermodel for the entailment (C). By iterated application of the construction in the proof of Theorem 1 there exists t', agreeing with t on F, such that $(s,t',h) \in [\![T_1 \wedge \cdots \wedge T_k]\!]$. Thus we have $(s,t',h) \in [\![T_1 \wedge \cdots \wedge T_k \wedge \forall x_1 \ldots x_n\{P_A\}\ F\ \mathit{vs}\ \{Q_A\}]\!]$. But we also have $(s,t',h) \notin [\![\{P_B\}\ F\ \mathit{vs}\ \{Q_B\}]\!]$ and this suffices.

For 3.: We use the Crossing Out property to derive

$$\begin{array}{ll} & T_1 \wedge \cdots \wedge T_k\ \wedge\ \forall x_1 \ldots x_n\ \{T'_1 \wedge \cdots \wedge T'_r \wedge P_A\}\ F\ \mathit{ts}\ \{Q_A\} \\ \models & T'_1 \wedge \cdots \wedge T'_r\ \wedge\ \forall x_1 \ldots x_n\ \{T'_1 \wedge \cdots \wedge T'_r \wedge P_A\}\ F\ \mathit{ts}\ \{Q_A\} \\ \models & \forall x_1 \ldots x_n\ \{P_A\}\ F\ \mathit{ts}\ \{Q_A\}\ \ \models\ \ \{P_B\}\ F\ \mathit{vs}\ \{Q_B\} \end{array} \qquad \square$$

Corollary 3 Correctness of the procedure DECIDE-ENT-1 *(Fig. 4). The procedure* DECIDE-ENT-1 *decides entailments $T_1 \wedge \cdots \wedge T_k \models T$ where $T_1,\ldots,T_k$ are level 1 SSH-triples about distinct procedures and T is a level 0 SSH-triple.*

Sketch proof. The first thing we must show is that $T_1 \wedge \cdots \wedge T_k \models T$ iff $T_1 \wedge \cdots \wedge T_k \models \text{FRESHEN}(T)$; this is a standard result. Then we need to prove that the entailment fails (with convenient counterexamples) if none of the triples $T_1,\ldots,T_k$ is about F; this is straightforward.

Next, we consider the call to INSTANTIATE. If this returns None, we can show that the entailment fails: Lemma 5 supplies the main part of the argument. On the other hand if an instantiation is returned we use Theorem 2 to show that it is *the* correct instantiation.

To finish, we use Theorem 3. If one of the calls to DECIDE-ENT-1 returns false, then part 1 of Theorem 3 applies; if all those calls return true, then parts 2 and 3 of Theorem 3 reduce the problem to a level 0 entailment problem, which Corollary 2 tells us is correctly solved. □

5 Conclusions

Revisiting our example. To demonstrate the working of our decision procedure, we revisit the verification condition (1) (on page 5) which arose from the example program in our introduction. Running on (1), DECIDE-ENT-1 first invokes FRESHEN which rewrites the desired conclusion to $\{\underline{\mathsf{D}} \star \mathit{ctr} \mapsto \{\!|\mathit{cnt} : \_|\!\}\}\ \mathit{SendToAll}\ z\ \{\underline{\mathsf{D}} \star \mathit{ctr} \mapsto \{\!|\mathit{cnt} : \_|\!\}\}$. Then the third triple on the left of $\models$ is chosen and INSTANTIATE determines that a should be instantiated with z. Now two recursive calls are made to DECIDE-ENT-1, to check the two entailments

$$\begin{array}{ll} & \forall x\ \{\underline{\mathsf{D}} \star \mathit{ctr} \mapsto \{\!|\mathit{cnt} : \_|\!\}\}\ \mathit{OldSend}\ x\ \{\underline{\mathsf{D}} \star \mathit{ctr} \mapsto \{\!|\mathit{cnt} : \_|\!\}\} \\ \models & \forall x\ \{\underline{\mathsf{D}} \star \mathit{ctr} \mapsto \{\!|\mathit{cnt} : \_|\!\}\}\ \mathit{OldSend}\ x\ \{\underline{\mathsf{D}} \star \mathit{ctr} \mapsto \{\!|\mathit{cnt} : \_|\!\}\} \end{array}$$

$$\begin{array}{ll} & \forall x\ \{\underline{\mathsf{D}}\}\ \mathit{NewUserProc}\ x\ \{\underline{\mathsf{D}}\} \\ \models & \forall x\ \{\underline{\mathsf{D}} \star \mathit{ctr} \mapsto \{\!|\mathit{cnt} : \_|\!\}\}\ \mathit{NewUserProc}\ z\ \{\underline{\mathsf{D}} \star \mathit{ctr} \mapsto \{\!|\mathit{cnt} : \_|\!\}\} \end{array}$$

We trace into the second of these which, after dealing with the quantifiers, makes a call

$$\text{DECIDE-ENT-QF-0}\left(\begin{array}{l}\{\underline{\mathrm{D}}\}\ \mathit{NewUserProc}\ z\ \{\underline{\mathrm{D}}\},\\ \{\underline{\mathrm{D}}\star \mathit{ctr}\mapsto \{\!|\mathit{cnt}:\_|\!\}\}\ \mathit{NewUserProc}\ z\ \{\underline{\mathrm{D}}\star \mathit{ctr}\mapsto\{\!|\mathit{cnt}:\_|\!\}\}\end{array}\right)$$

Inside this call, $R = \underline{\mathrm{D}}\star \mathit{ctr}\mapsto\{\!|\mathit{cnt}:\_|\!\} -_{\mathrm{SSH}} \underline{\mathrm{D}}$ is computed and found to be $\mathit{ctr}\mapsto\{\!|\mathit{cnt}:\_|\!\}$. Then the algorithm checks that $R\star\underline{\mathrm{D}}$ ensures $\underline{\mathrm{D}}\star \mathit{ctr}\mapsto\{\!|\mathit{cnt}:\_|\!\}$ which is the case. Returning to the outer call to DECIDE-ENT-1, the final check is

$$\text{DECIDE-ENT-QF-0}\left(\begin{array}{l}\{\underline{\mathrm{D}}\star \mathit{ctr}\mapsto \{\!|\mathit{cnt}:\_|\!\}\}\ \mathit{SendToAll}\ z\ \{\underline{\mathrm{D}}\star \mathit{ctr}\mapsto\{\!|\mathit{cnt}:\_|\!\}\},\\ \{\underline{\mathrm{D}}\star \mathit{ctr}\mapsto \{\!|\mathit{cnt}:\_|\!\}\}\ \mathit{SendToAll}\ z\ \{\underline{\mathrm{D}}\star \mathit{ctr}\mapsto\{\!|\mathit{cnt}:\_|\!\}\}\end{array}\right)$$

Future work. A natural next step for this line of work is to investigate which of the restrictions we have made are necessary to obtain decidability. For instance, it appears that our method will generalise to arbitrary nesting depth if we can handle conjunctions $T_1\wedge\cdots\wedge T_k$ containing multiple triples about the same procedure. Of particular interest is to see whether one can allow, without breaking decidability, a limited use of a universal quantifier $\underline{\forall}$ over assertion variables, which allows much more generic specifications. The specification we gave for APPLY2 in Fig. 2 is generic with respect to the parameters F, G and a, but *not* with respect to the invariant that F and G must preserve, which is fixed as $\underline{\mathrm{D}}\star \mathit{ctr}\mapsto\{\!|\mathit{cnt}:\_|\!\}$. The assertion variable $\underline{\mathrm{D}}$ represents an arbitrary invariant, but it is the *same* arbitrary invariant as mentioned in the specification of MAIN. This suffices for our example, but a better specification would be:

$$\underline{\forall}\mathrm{I}\forall F\forall G\forall a\ \Big\{\underline{\mathrm{I}}\wedge\forall x\{\underline{\mathrm{I}}\}\ F\ x\ \{\underline{\mathrm{I}}\}\ \wedge\ \forall x\{\underline{\mathrm{I}}\}\ G\ x\ \{\underline{\mathrm{I}}\}\Big\}\ \text{APPLY2}\ F\ G\ a\ \{\underline{\mathrm{I}}\}$$

One sees here that $\underline{\forall}$ supports reasoning akin to that provided by the *hypothetical frame rule* described in [OYR04], but also goes beyond it: an even more general specification is

$$\underline{\forall}\mathrm{I}\,\underline{\forall}\mathrm{J}\,\underline{\forall}\mathrm{K}\forall F\forall G\forall a\ \Big\{\underline{\mathrm{I}}\wedge\forall x\{\underline{\mathrm{I}}\}\ F\ x\ \{\underline{\mathrm{J}}\}\ \wedge\ \forall x\{\underline{\mathrm{J}}\}\ G\ x\ \{\underline{\mathrm{K}}\}\Big\}\ \text{APPLY2}\ F\ G\ a\ \{\underline{\mathrm{K}}\}$$

Universally quantified assertion variables can be instantiated with arbitrary formulae — we have $\underline{\forall}\mathrm{I}\Phi\models\Phi[\underline{\mathrm{I}}\backslash\Psi]$ — but the issue is how one computes the "right" Ψ. Finally, we plan to extend our work to treat programs where procedures are stored on the heap as well as on the stack.

Acknowledgements: This work was supported by EPSRC grant EP/G003173/1. We thank the anonymous referees for their suggestions for improving the presentation.

Bibliography

[BCO05a] J. Berdine, C. Calcagno, P. W. O'Hearn. Smallfoot: Modular Automatic Assertion Checking with Separation Logic. In *FMCO*. Pp. 115–137. 2005.

[BCO05b] J. Berdine, C. Calcagno, P. W. O'Hearn. Symbolic Execution with Separation Logic. In *APLAS*. Pp. 52–68. 2005.

[BIP08] M. Bozga, R. Iosif, S. Perarnau. Quantitative Separation Logic and Programs with Lists. In *IJCAR*. Pp. 34–49. 2008.

[BRSY08] L. Birkedal, B. Reus, J. Schwinghammer, H. Yang. A Simple Model of Separation Logic for Higher-Order Store. In *ICALP (2)*. Pp. 348–360. 2008.

[BTY06] L. Birkedal, N. Torp-Smith, H. Yang. Semantics of Separation-Logic Typing and Higher-order Frame Rules for Algol-like Languages. *LMCS* 2(5), 2006.

[BY07] L. Birkedal, H. Yang. Relational Parametricity and Separation Logic. In *FoSSaCS*. Pp. 93–107. 2007.

[FLL$^+$02] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, R. Stata. Extended Static Checking for Java. In *PLDI*. Pp. 234–245. 2002.

[HYB05] K. Honda, N. Yoshida, M. Berger. An Observationally Complete Program Logic for Imperative Higher-Order Functions. In *LICS*. Pp. 270–279. 2005.

[KAB$^+$09] N. R. Krishnaswami, J. Aldrich, L. Birkedal, K. Svendsen, A. Buisse. Design patterns in separation logic. In *TLDI*. Pp. 105–116. 2009.

[MN05] F. Mehta, T. Nipkow. Proving pointer programs in higher-order logic. *Inf. Comput.* 199(1-2):200–227, 2005.

[NDQC07] H. H. Nguyen, C. David, S. Qin, W.-N. Chin. Automated Verification of Shape and Size Properties Via Separation Logic. In *VMCAI*. Pp. 251–266. 2007.

[NMB08] A. Nanevski, J. G. Morrisett, L. Birkedal. Hoare type theory, polymorphism and separation. *J. Funct. Program.* 18(5-6):865–911, 2008.

[OYR04] P. W. O'Hearn, H. Yang, J. C. Reynolds. Separation and information hiding. In *POPL*. Pp. 268–280. 2004.

[Rey02] J. C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*. Pp. 55–74. 2002.

[RS06] B. Reus, J. Schwinghammer. Separation Logic for Higher-Order Store. In *CSL*. Pp. 575–590. 2006.

[SBRY09] J. Schwinghammer, L. Birkedal, B. Reus, H. Yang. Nested Hoare Triples and Frame Rules for Higher-Order Store. In *CSL*. Pp. 440–454. 2009.

[ZKR08] K. Zee, V. Kuncak, M. C. Rinard. Full functional verification of linked data structures. In *PLDI*. Pp. 349–361. 2008.