

Verification Condition Generation for Conditional Information Flow*

Torben Amtoft[†]
Kansas State University
Manhattan, KS, USA
tamtoft@cis.ksu.edu

Anindya Banerjee[‡]
Kansas State University
Manhattan, KS, USA
ab@cis.ksu.edu

August 19, 2007

Abstract

We formulate an intraprocedural information flow analysis algorithm for sequential, heap manipulating programs. We prove correctness of the algorithm, and argue that it can be used to verify some naturally occurring examples in which information flow is conditional on some Hoare-like state predicates being satisfied. Because the correctness of information flow analysis is typically formulated in terms of noninterference of pairs of computations, the algorithm takes as input a program together with two-state assertions as postcondition, and generates two-state preconditions together with verification conditions. To process heap manipulations and while loops, the algorithm must additionally be supplied “object flow invariants” as well as “loop flow invariants” which are themselves two-state, and possibly conditional.

1 Introduction

Information flow analyses are used to ensure that programs satisfy confidentiality policies. Such policies are expressed by labeling variables with security levels, e.g., H for secret/classified and L for public/observable/unclassified. For a given policy, a program P satisfies *noninterference* (NI) [18] provided that for any *two* runs of P , if P is executed from two input states that are L -indistinguishable (i.e., the input states agree on the values of L -variables) then it yields output states that are also L -indistinguishable. A sound information flow analysis guarantees that the programs it accepts are noninterferent.

This paper formulates a sound intraprocedural information flow analysis *algorithm* — rather than a type-based or logic-based *specification* — for heap manipulating programs. We assume that such programs are more or less decorated with assertion statements and loop/object invariants; such invariants can in many cases be automatically checked by tools such as BLAST [20], ESC/Java [13] or Spec# [7]. A novel aspect of the algorithm is that it reasons about possibly *conditional* information flow, and also handles while loops and common data structures when armed with *flow invariants* (introduced in the sequel). We leave the automatic *inference* of flow invariants for future work.

*Technical Report, KSU CIS-TR-2007-2.

[†]Supported in part by a grant from AFOSR, and by a grant from Rockwell Collins.

[‡]Supported in part by *Advanced Programming Tools* group, IBM T. J. Watson Research Center, by an AFOSR grant, and by NSF grants CNS-0627748, CCR-0296182, ITR-0326577.

Given a variable x labeled L , the formulation of noninterference entails that we restrict our attention to pairs of states σ_1, σ_2 where $\sigma_1(x) = \sigma_2(x)$. This observation inspired Amtoft et al. [2, 1] to a logical rendition of NI which uses *agreement* assertions of the form $x \bowtie$, where two states σ_1, σ_2 satisfy $x \bowtie$ when $\sigma_1(x) = \sigma_2(x)$. If a program P has observable input variables x_1, \dots, x_n , and observable output variables y_1, \dots, y_m , then NI can be recast as

$$\{x_1 \bowtie \wedge \dots \wedge x_n \bowtie\} P \{y_1 \bowtie \wedge \dots \wedge y_m \bowtie\}.$$

The meaning (partial correctness) of the above triple is that for any two states σ_1, σ_2 that agree on the values of x_1, \dots, x_n (as asserted by the precondition), if one run of P transforms σ_1 to σ'_1 and another run of P transforms σ_2 to σ'_2 , then the values of y_1, \dots, y_m agree in the final states, σ'_1, σ'_2 (as asserted by the postcondition).¹

Amtoft et al. [1] specify, in logical form, a modular interprocedural information flow analysis for sequential, heap-manipulating programs. The specification is flow sensitive (unlike most type-based approaches), and uses “region assertions” to associate variables with “abstract locations”, abstracting sets of concrete locations. Therefore it is possible to do a limited form of alias analysis, and we give examples where the absence of information leaks is demonstrated by showing that two variables must not alias; we also address applications to analyzing observational purity. Moreover, the specification can be used to check compliance with delimited release policies [23] in a technically straightforward manner: extend agreements over variables to agreements over “escape-hatch” expressions that syntactically specify such policies. More recently, the specification has been proposed as a crucial component for the verification of state-dependent declassification policies [5].

Amtoft et al. [1] present an analysis algorithm which, however, has some limitations: it essentially needs to know the “shape” of the heap, and must be provided method summaries. While working to mitigate those limitations, we discovered some shortcomings of the underlying logic, primarily due to the fact that it employs *three* kinds of primitive assertions (agreement, region, programmer) which can be combined only through conjunction. As a consequence, programmer assertions are not smoothly integrated, and it is not possible to capture *conditional* information flows. These shortcomings make it difficult to analyze information flow in non-trivial programs, especially ones that involve reasoning about common data structures. (A similar situation prevails with extant security type systems [25, 4, 21].)

Contributions. This paper shows how to reason about information flow that may be conditional, and how to compute it for programs that may manipulate common data structures. The algorithm (Sect. 4) takes as input a program and a (possibly conditional) agreement assertion as postcondition, and as output generates preconditions and verification conditions (VCs). Currently, the algorithm expects the user to provide loop invariants and object invariants that are themselves (conditional) agreement assertions; we call such invariants *flow invariants*. The algorithm always terminates, but the VCs may be unsatisfiable; this will happen if the flow invariants are not strong enough. The correctness of the algorithm is proved directly wrt. the underlying semantics; this is unlike [2, 1] which first establish the semantic soundness of a logic and next provide a sound implementation of that logic. We use the algorithm to verify some naturally occurring examples. A prototype implementation² is currently being developed by Jonathan Hoag.

¹Two remarks: (a) The connection with NI based on security labels [25] is that for any well-labeled program, P , if l_1, \dots, l_n are all the L -variables in P then $l_1 \bowtie \wedge \dots \wedge l_n \bowtie$ is an invariant. (b) To model security lattices with more than two elements, say $L \leq M \leq H$, multiple specifications are needed, like “if input states agree on L then output states agree on L ” and “if input states agree on L, M then output states agree on L, M ”.

²Available at <http://people.cis.ksu.edu/~jch5588/securityflow/SecurityFlow.html>. It requires Java 1.5.11. As of writing, it handles assignments, conditionals, and while loops.

An example loop flow invariant is $x \times$, with the following informal semantics: if two states, σ_1 and σ_2 , agree on the value of x , and one iteration of the loop transforms σ_1 into σ'_1 and σ_2 into σ'_2 , then also σ'_1 and σ'_2 agree on the value of x . If the invariant is conditional, like $i > n \Rightarrow x \times$, then σ'_1 and σ'_2 are required to agree on x only if they both assert $i > n$, whereas σ_1 and σ_2 can be assumed to agree on x only if they both assert $i > n$. (We defer examples of object flow invariants to Sect. 2.) A second contribution of the paper is the underlying semantic framework (Sect. 3) for such conditional assertions that mixes ordinary, Hoare-logic style predicates with two-state agreement assertions.

A third contribution is the smooth integration with standard assertions, the presence of which can help the algorithm to increase precision. A simple example of this is the program

```
if w then x := 7 else x := 7; assert(x = 7).
```

Given the postcondition $x \times$, the algorithm will compute $(x = 7 \Rightarrow x \times)$ as the precondition of the assertion statement; this is justified in all contexts because we employ a correctness criterion which considers only executions that terminate successfully, and the assertion will abort if $x \neq 7$ (which of course cannot happen in the given context). Since $(x = 7) \Rightarrow x \times$ always holds, it can be simplified to *true*, which when given as postcondition to the conditional is also returned as the precondition. Without the ability to use and/or derive/infer the assertion statement, however, the precondition would need to include $w \times$. The inference of such “standard” assertions can be done by, e.g., BLAST, but will not be our concern in this paper.

2 Examples

We now illustrate, by way of examples in Figs. 1 and 2, the issues involved in verifying information flow policies for while loops, as well as for programs that manipulate the heap using field update, field access and object allocation. Claims about the behavior of our algorithm will be substantiated in Sect. 5.

Loop flow invariants. Consider the program P in Fig. 1(a), and the policy specification $\{x \times\} - \{res \times\}$. Does P satisfy this specification? That is, will two runs of P for which the values of x agree in the initial states also yield final states in which the values of res agree? Note that the precondition does not make any commitments about $v \times$ and $h \times$.

To answer the above question, observe that since the program updates res (line 4), for $res \times$ to hold at the end, $v \times$ must also hold. Alas, $v \times$ holds only at the beginning of every *odd* iteration of the loop — but fortunately, this is exactly when v is used to update res . It turns out that to verify the program we need the loop flow invariant $odd(i) \Rightarrow v \times$ which testifies to *conditionally secure information flow* within the loop.³ Furthermore, after res is updated, the assignment to v (line 5) *invalidates* the invariant because $h \times$ may not hold. But because i is incremented by 1 (line 8), $odd(i)$ is falsified and the invariant is reestablished, vacuously, at the beginning of the next (even) iteration of the loop. Our algorithm, applied to the program in Fig. 1(a) and equipped with a loop flow invariant containing $odd(i) \Rightarrow v \times$, generates valid verification conditions (VCs) together with a precondition that includes $x \times$ but *not* $h \times$. Thus the program is deemed secure.

Note that standard security type systems do not take *conditional* loop flow invariants like the one above into account and therefore, given that res has type L and h has type H , reject the program as insecure: well-typedness demands v to have type L , due to the assignment to res (line 4), and also to have type H , due to the assignment to v (line 5). (The security type given to a while loop can be interpreted as an unconditional loop flow invariant, which in this case is not precise enough.)

³Note that we do not want $odd(i)$ in the precondition along with $x \times$; i can be any integer, odd or even.

<pre> 1. $i := 0; v := 0; res := 0;$ 2. while ($i < 7$) do 3. if $odd(i)$ 4. then $res := res + v;$ 5. $v := v + h;$ 6. else $v := x;$ 7. fi; 8. $i := i + 1;$ 9. od </pre> <p style="text-align: center;">(a)</p>	<pre> 1. open x in 2. $y := .src;$ 3. $i := .idx;$ 4. close; 5. open y in 6. assert ($odd(i) \rightarrow odd(.idx)$); 7. $q := .val;$ 8. close; 9. open x in 10. assert ($.idx = i$); 11. $.val := q;$ 12. $res := .val;$ 13. close; </pre> <p style="text-align: center;">(b)</p>
--	--

Figure 1: Two examples that illustrate **(a)** loop flow invariants, and **(b)** object flow invariants and scoped heap operations. $odd(i)$ is expressible as $(i \bmod 2 = 1)$ in our language.

Object flow invariants. The next example is motivated by an actual program, used in hardware verification of operational amplifiers, that was provided by our industrial collaborators, Rockwell-Collins. The example also serves to introduce the heap manipulating fragment of the language we analyze. We are given a collection of objects where each object has three fields: val containing its “value”, src containing the “source” object whose value will be used to update the val field, and idx containing the object’s index in the collection. The overall policy specification is that odd elements should be public; formally, we need to specify

$$\begin{aligned}
odd(o.idx) &\Rightarrow (o.val) \times \text{ and} \\
odd(o.idx) &\Rightarrow (o.src) \times.
\end{aligned}$$

Given this *object flow invariant*, we now ask whether the program

```

 $y := x.src; i := x.idx;$ 
 $q := y.val; x.val := q; res := x.val$ 

```

satisfies the policy $\{x \times\} - \{odd(i) \Rightarrow res \times\}$.

Intuitively, for this to hold we must demand that if the val field of an object with odd index is updated with a value q , then the source object whose val field contains q must be one with odd index. We therefore assert an implication based on the above intuition:

```

 $y := x.src; i := x.idx;$ 
assert ( $odd(i) \rightarrow odd(y.idx)$ );
 $q := y.val; x.val := q; res := x.val$ 

```

It is well-known that standard Hoare logic does not handle heaps very well, a key issue being “pointer swing” that leads to aliasing. An update of $u.f$ may affect $w.f$ if u and w may alias. Rather than employ a may-alias

analysis, we demand that all field accesses and updates be *scoped*. For example, a field access, $y := x.f$, occurs as **open** x **in** $y := .f$; **close**. A field update, $x.f := y$, occurs as **open** x **in** $.f := y$; **close**.

Fig. 1(b) shows the program that corresponds to the one above. It also exemplifies the syntax of the language that we analyze: it is a simple imperative language, extended with assertions and scoped heap manipulating commands (field accesses, field updates, object allocation). A formal BNF appears in Sect. 3.

Because of scoped field accesses and updates, we no longer need a prefix for a field as this is clear from the scope. In general, to compare claims about two different scopes, as in $\text{assert}(\text{odd}(x.idx) \rightarrow \text{odd}(y.idx))$, we need to save the result of $x.idx$ into a variable i . It turns out that we must assist our analysis by explicitly asserting (line 10) that when x is opened the second time, the index is still i .

The task of each scope is now to maintain the object flow invariant. To see that reasoning about aliasing is not a problem, observe that it is possible that updating the object pointed to by x also updates the object pointed to by y . However, this is permissible as long as the new object state satisfies the object flow invariant.

Note that the assertions used in the program (lines 6, 10) might be eliminated by theorem proving tools used in conjunction with other static analyses. In particular, the first assertion (line 6) could be eliminated in case we can prove, say, that for all objects o we have $o.src.idx = o.idx + 2$.

Our algorithm for verification condition generation, when given as input the program in Fig. 1(b), postcondition $\text{odd}(i) \Rightarrow res \times$, and object flow invariant $\{\text{odd}(.idx) \Rightarrow .val \times \wedge \text{odd}(.idx) \Rightarrow .src \times\}$, generates valid VCs, and the precondition $true \Rightarrow x \times$ (equivalent to $x \times$).

Combining loop flow invariants, object flow invariants, and allocation. Next, we consider the example in Fig. 2, featuring a heterogeneous list pointed to by x and represented as a node chain, where one node can be reached from another by traversing *next* links. The *val* field of each node contains either a high (*H*) value or a low (*L*) value, where the protocol is that a value is *L* provided it is less than 10. Informally, the list satisfies an object flow invariant $.val < 10 \Rightarrow val \times$.

We wish to split the list pointed to by x and output two homogeneous lists, pointed to by y and z ; here y will point to a list containing all the nodes of x with *val* fields that are *L*, i.e., less than 10, whereas z will point to a list containing the other nodes of x . Since the final value of *res* is taken from the list pointed to by y , the overall policy specification is $\{x \times\} - \{res \times\}$. Our algorithm verifies that the program in Fig. 2 satisfies this specification: from postcondition $res \times$ it generates precondition $x \times$ and some valid VCs.

For the verification process, object flow invariants are needed; one might think that we need one invariant for each kind of node but those can be combined into a “universal” object flow invariant, using a field *.t* which tags the lists x , y and z with 1, 2, 3 respectively.

$$\begin{aligned} (.t = 1 \wedge .val < 10) &\Rightarrow .val \times \\ .t = 1 &\Rightarrow (.val < 10) \times & .t = 1 &\Rightarrow .next \times \\ .t = 2 &\Rightarrow .val \times & .t = 2 &\Rightarrow .next \times \end{aligned}$$

Here $(.val < 10) \times$ is satisfied by a pair of states if they agree on the value of the comparison (but not necessarily on the value of *.val*).

The example also shows scoped object allocation, where new objects (pointed to by y_1 and z_1) are allocated in the heap and their fields initialized as shown. Once all fields are initialized, the object flow invariant must have been established so that when the scope **new** . . . **close** is exited the object is in a “steady state”.

Readers familiar with the Boogie methodology [6] might notice some similarity between **open** . . . **close** and Boogie’s **unpack** and **pack**, where the object invariant must be reestablished at the end of every field update. Boogie requires object invariants to be associated with every object of a class. Our language seems

<ol style="list-style-type: none"> 1. $y := nil; z := nil;$ 2. while $x \neq nil$ do 3. open x in assert($.t = 1$); $v := .val; n := .next;$ close; 4. $x := n;$ 5. if $v < 10$ 6. then new y_1 in $.val := v; .next := y; .t := 2;$ close; 7. $y := y_1;$ 8. else new z_1 in $.val := v; .next := z; .t := 3;$ close; 9. $z := z_1;$ 10. fi; 11. od; 	<ol style="list-style-type: none"> 12. $res := nil;$ 13. while $y \neq nil$ do 14. open y in 15. assert($.t = 2$); 16. $res := .val;$ 17. $y := .next;$ 18. close; 19. od
---	--

Figure 2: List splitting

impoverished in comparison to Boogie’s in that we have the equivalent of a single universal class, but as the above object flow invariant shows, the use of tags enables us to encode multiple invariants.

3 Syntax and Semantics

Expression syntax. An expression $E \in \mathbf{Exp}$ is either an arithmetic expression $A \in \mathbf{AExp}$ or a boolean expression $B \in \mathbf{BExp}$, given by the syntax

$$\begin{aligned}
 A &::= x \mid .f \mid c \mid nil \mid A \text{ op } A \\
 B &::= A \text{ bop } A
 \end{aligned}$$

where we use x, y, \dots to range over variables in \mathbf{Var} , and f, g, \dots to range over field names in \mathbf{Fld} , and c to range over integer constants, and op to range over arithmetic operators in $\{+, \times, \text{mod}, \dots\}$, and bop to range over comparison operators in $\{=, <, \dots\}$.

We write $\text{fv}(E)$ (or $\text{ff}(E)$) for the variables (field names) occurring free in E . We write $E[A/x]$ for the result of substituting all occurrences of x in E by A ; similarly we write $E[A/.f]$. We say that E is *field-free* if E contains no field names, and that E is an *object expression* if E contains no variables.

We assume that each variable and each field is either for integers or for pointers (to objects), as prescribed by a function type mapping $\mathbf{Var} \cup \mathbf{Fld}$ into $\{\text{int}, \text{obj}\}$. We shall only consider programs that are “well-typed” in that respect. In particular, we disallow pointer arithmetic; the only operation allowed on pointers is pointer equality. Thus we have

Fact 3.1 *Assume that $\text{type}(x) = \text{obj}$. Then $x \in \text{fv}(A)$ only if A is syntactically equal to x , and $x \in \text{fv}(B)$ only if B is of the form either $(x = x)$ or $(A = x)$ or $(x = A)$ with $x \notin \text{fv}(A)$.*

Semantic domains. A value ($v \in \mathbf{Val}$) is an integer n , a location $\ell \in \mathbf{Loc}$, or nil ; default values (which are also arithmetic expressions) are defined as $\text{deflt}(\text{int}) = 0$ and $\text{deflt}(\text{obj}) = nil$, and we write $\text{deflt}(f)$ for $\text{deflt}(\text{type}(f))$. A *store* $s \in \mathbf{Store}$ maps variables to values, an *object state* r maps field names to values, and a *heap* $h \in \mathbf{Heap}$ maps locations to object states; the notions of $\text{dom}(_)$ and $\text{ran}(_)$ are as usual except that (with misuse of notation) we write $\text{ran}(h) = \{v \mid \exists \ell \in \text{dom}(h), f \in \mathbf{Fld} : v = h(\ell)(f)\}$. We write

	$RS ::= \text{skip}$		$TS ::= \text{skip}$
assertion	$\text{assert}(\phi)$		$\text{assert}(\phi)$
sequential execution	$RS ; RS$		$TS ; TS$
conditional	$\text{if } B \text{ then } RS \text{ else } RS$		$\text{if } B \text{ then } TS \text{ else } TS$
iteration	$\text{while } B \text{ do } RS$		$\text{while } B \text{ do } TS$
variable assignment	$x := A$		$x := A$
field update	$.f := A$		
object allocation			$\text{new } x \text{ in } RS \text{ close}$
object manipulation			$\text{open } x \text{ in } RS \text{ close}$

Figure 3: Command syntax. For TS , all instances of A , B and ϕ must be field-free.

$[s \mid x \mapsto v]$ for the store that is like s except that it maps x into v ; similarly we write $[r \mid f \mapsto v]$ and $[h \mid \ell \mapsto r]$.

Expression semantics. The semantics of an arithmetic (boolean) expression is a function from stores and object states into values (booleans). If an expression E is field-free (an object expression), the “ r ” component (the “ s ” component) can be omitted.

$$\begin{aligned} \llbracket x \rrbracket_r^s &= s(x), & \llbracket .f \rrbracket_r^s &= r(f), & \llbracket c \rrbracket_r^s &= c, & \llbracket \text{nil} \rrbracket_r^s &= \text{nil} \\ \llbracket A_1 + A_2 \rrbracket_r^s &= \llbracket A_1 \rrbracket_r^s + \llbracket A_2 \rrbracket_r^s, \text{ etc.} \\ \llbracket A_1 < A_2 \rrbracket_r^s &= \text{True iff } \llbracket A_1 \rrbracket_r^s < \llbracket A_2 \rrbracket_r^s, \text{ etc.} \end{aligned}$$

One-state assertions. We use $\phi \in \mathbf{1Assert}$ to range over “standard” assertions, given by the syntax

$$\phi ::= B \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg \phi$$

We may define *true* as $0 = 0$, and *false* as $0 = 1$; as usual, we define $\phi_1 \rightarrow \phi_2$ as $\neg \phi_1 \vee \phi_2$. We write $\phi[A/x]$ for the result of substituting all occurrences of x in ϕ by A ; similarly we define $\phi[A/.f]$.

The satisfaction relation for assertions reads $s, r \models \phi$ and denotes that ϕ holds in the *one state* comprised of the store s and the object state r . The definition is inductive in ϕ : $s, r \models B$ iff $\llbracket B \rrbracket_r^s = \text{True}$; $s, r \models \phi_1 \wedge \phi_2$ iff $s, r \models \phi_1$ and $s, r \models \phi_2$, etc. We say that ϕ is field-free if ϕ contains no field names, in which case the r component can be omitted; we say that ϕ is an *object assertion* if ϕ contains no variables, in which case the s component can be omitted.

Command syntax. A command $S \in \mathbf{Cmd}$ is either a *top-level command* $TS \in \mathbf{TCmd}$ or a *record command* $RS \in \mathbf{RCmd}$; the latter is executed within the scope of a *single* object and is thus allowed to reference its fields. The syntax is given in Fig. 3, where in the grammar for TS we demand that all instances of A , B , and ϕ are field-free.

Command semantics. A record command transforms the store, and the state of the object being manipulated, into another store and another object state; hence its semantics is given in relational style, in the form $s, r \llbracket RS \rrbracket s', r'$. A top-level command transforms a store and a heap into another store and another heap;

$$\begin{aligned}
s, r \llbracket \text{skip} \rrbracket s', r' & \text{ iff } s' = s \text{ and } r' = r \\
s, h \llbracket \text{skip} \rrbracket s', h' & \text{ iff } s' = s \text{ and } h' = h \\
s, r \llbracket \text{assert}(\phi) \rrbracket s', r' & \text{ iff } s, r \models \phi \text{ and } s' = s \text{ and } r' = r \\
s, h \llbracket \text{assert}(\phi) \rrbracket s', h' & \text{ iff } s \models \phi \text{ and } s' = s \text{ and } h' = h \\
s, r \llbracket RS_1 ; RS_2 \rrbracket s', r' & \text{ iff } \exists s'', r'' : s, r \llbracket RS_1 \rrbracket s'', r'' \text{ and } s'', r'' \llbracket RS_2 \rrbracket s', r' \\
s, h \llbracket TS_1 ; TS_2 \rrbracket s', h' & \text{ iff } \exists s'', h'' : s, h \llbracket TS_1 \rrbracket s'', h'' \text{ and } s'', h'' \llbracket TS_2 \rrbracket s', h' \\
s, r \llbracket \text{if } B \text{ then } RS_1 \text{ else } RS_2 \rrbracket s', r' & \text{ iff } (\llbracket B \rrbracket_r^s = \text{True} \text{ and } s, r \llbracket RS_1 \rrbracket s', r') \\
& \text{ or } (\llbracket B \rrbracket_r^s = \text{False} \text{ and } s, r \llbracket RS_2 \rrbracket s', r') \\
s, h \llbracket \text{if } B \text{ then } TS_1 \text{ else } TS_2 \rrbracket s', h' & \text{ iff } (\llbracket B \rrbracket^s = \text{True} \text{ and } s, h \llbracket TS_1 \rrbracket s', h') \\
& \text{ or } (\llbracket B \rrbracket^s = \text{False} \text{ and } s, h \llbracket TS_2 \rrbracket s', h') \\
s, r \llbracket x := A \rrbracket s', r' & \text{ iff } \exists v : v = \llbracket A \rrbracket_r^s \text{ and } s' = [s \mid x \mapsto v] \text{ and } r' = r \\
s, h \llbracket x := A \rrbracket s', h' & \text{ iff } \exists v : v = \llbracket A \rrbracket^s \text{ and } s' = [s \mid x \mapsto v] \text{ and } h' = h \\
s, r \llbracket .f := A \rrbracket s', r' & \text{ iff } \exists v : v = \llbracket A \rrbracket_r^s \text{ and } s' = s \text{ and } r' = [r \mid f \mapsto v] \\
s, h \llbracket \text{new } x \text{ in } RS \text{ close} \rrbracket s', h' & \text{ iff } \exists l, r_0, r' : (l \notin \text{dom}(h) \cup \text{ran}(h) \cup \text{ran}(s) \text{ and } r_0 = \text{deflt} \\
& \text{ and } [s \mid x \mapsto l], r_0 \llbracket RS \rrbracket s', r' \text{ and } h' = [h \mid l \mapsto r']) \\
s, h \llbracket \text{open } x \text{ in } RS \text{ close} \rrbracket s', h' & \text{ iff } \exists l, r, r' : (l = s(x) \text{ and } r = h(l) \\
& \text{ and } s, r \llbracket RS \rrbracket s', r' \text{ and } h' = [h \mid l \mapsto r']) \\
s, r \llbracket \text{while } B \text{ do } RS \rrbracket s', r' & \text{ iff } \exists i \geq 0 : s, r f_i s', r' \text{ where } f_i \text{ is inductively defined by:} \\
& \quad s, r f_0 s', r' \text{ iff } \llbracket B \rrbracket_r^s = \text{False} \text{ and } s' = s \text{ and } r' = r \\
& \quad s, r f_{i+1} s', r' \text{ iff } \exists s'', r'' : (\llbracket B \rrbracket_r^s = \text{True} \text{ and} \\
& \quad \quad s, r \llbracket RS \rrbracket s'', r'' \text{ and } s'', r'' f_i s', r') \\
s, h \llbracket \text{while } B \text{ do } TS \rrbracket s', h' & \text{ iff } \exists i \geq 0 : s, h f_i s', h' \text{ where } f_i \text{ is inductively defined by:} \\
& \quad s, h f_0 s', h' \text{ iff } \llbracket B \rrbracket^s = \text{False} \text{ and } s' = s \text{ and } h' = h \\
& \quad s, h f_{i+1} s', h' \text{ iff } \exists s'', h'' : (\llbracket B \rrbracket^s = \text{True} \text{ and} \\
& \quad \quad s, h \llbracket TS \rrbracket s'', h'' \text{ and } s'', h'' f_i s', h')
\end{aligned}$$

Figure 4: Command semantics.

thus its semantics is given in the form $s, h \llbracket TS \rrbracket s', h'$. In Fig. 4, the semantics is defined inductively on RS and TS . Note that for some TS and s, h , there may not exist any s', h' such that $s, h \llbracket TS \rrbracket s', h'$ (modulo the choice of fresh location for object allocation, there exists at most one s', h'); this can happen if a **while** loop does not terminate, or an **assert** fails. Also note that if $s, h \llbracket TS \rrbracket s', h'$ then $\text{dom}(s) \subseteq \text{dom}(s')$ and $\text{dom}(h) \subseteq \text{dom}(h')$, and that if $s, r \llbracket RS \rrbracket s', r'$ then $\text{dom}(s) \subseteq \text{dom}(s')$.

Two-state assertions. We shall use $\theta \in \mathbf{2Assert}$ to range over conditional agreement assertions, also called *2-assertions*; they are of the form $\phi \Rightarrow E \times$ which intuitively is satisfied by a pair of states if either at least one of them does not satisfy ϕ , or they agree on the value of E . As we cannot expect two runs to choose the same fresh location for object allocation, we employ a bijection β between locations; we extend

β so that $c \beta c$ for all integers c , $\text{nil} \beta \text{nil}$, $\text{True} \beta \text{True}$, and $\text{False} \beta \text{False}$.

Then we define $s, r \& s_1, r_1 \models_{\beta} \theta$, the satisfaction relation for 2-assertions, by

$$s, r \& s_1, r_1 \models_{\beta} \phi \Rightarrow E \times \text{ iff whenever } s, r \models \phi \text{ and } s_1, r_1 \models \phi \text{ then } \llbracket E \rrbracket_r^s \beta \llbracket E \rrbracket_{r_1}^{s_1}.$$

For $\theta = (\phi \Rightarrow E \times)$, we call ϕ the antecedent of θ and write $\phi = \text{ant}(\theta)$, and we call E the consequent of θ and write $E = \text{con}(\theta)$. We say that θ is field-free if it contains no field names, in which case the r and r_1 can be omitted, and say that θ is an object assertion if it contains no variables, in which case the s and s_1 can be omitted. We often write $E \times$ for $\text{true} \Rightarrow E \times$.

We use $\Theta \in \mathcal{P}(\mathbf{2Assert})$ to range over sets of 2-assertions, with conjunction implicit. Thus

$$s, r \& s_1, r_1 \models_{\beta} \Theta \text{ iff } \forall \theta \in \Theta : s, r \& s_1, r_1 \models_{\beta} \theta.$$

Example 3.2 We might specify the behavior of an ATM using the 2-assertions

$$\{ \text{pin} = 1234 \Rightarrow \text{out} \times, \text{pin} \neq 1234 \Rightarrow \text{out} \times \}.$$

This allows out to depend on whether pin is 1234 or not, but *not* to depend on how “close” pin is to 1234. Note that this specification is *not* equivalent to $(\text{pin} = 1234 \vee \text{pin} \neq 1234) \Rightarrow \text{out} \times$ (which is just $\text{true} \Rightarrow \text{out} \times$).

Object flow invariants. We assume that there exists an object assertion \mathcal{I} that serves as a flow invariant for *every* object (cf. the discussion at the end of Sect. 2). We shall demand that for two runs of the program, the heap part obeys this invariant (except when an object is being manipulated within a scoped construct), and thus define

$$h \& h_1 \models_{\beta} \mathcal{I} \text{ iff for all } \ell, \ell_1 \text{ with } \ell \beta \ell_1 : h(\ell) \& h_1(\ell_1) \models_{\beta} \mathcal{I}.$$

4 Algorithm

We shall define, in Figs. 5 & 6, an algorithm VCgen for inferring preconditions, and verification conditions, from postconditions. We write⁴

$$[VC]\{\Theta\} (R) \Leftarrow S \{\Theta'\}$$

if from input S and Θ' , VCgen returns output Θ , R , and VC . Here S is a command, Θ' is the desired postcondition for S , and Θ is a precondition for S that is designed so as to be sufficient to establish Θ' ; if S is a top-level command then VCgen requires Θ' to be field-free and ensures that Θ is field-free. We shall shortly explain the role of the verification conditions VC , but shall first explain the R component which captures how 2-assertions in Θ relate to 2-assertions in Θ' . More precisely, we have $R \subseteq \Theta \times \{m, u\} \times \Theta'$ where tags m, u are mnemonics for “modified” and “unmodified”; we use γ to range over $\{m, u\}$. We write $\text{dom}(R) = \{\theta \mid \exists(\theta, -, -) \in R\}$ and $\text{ran}(R) = \{\theta' \mid \exists(-, -, \theta') \in R\}$. Intuitively, if $(\theta, -, \theta') \in R$ then θ is in the precondition because θ' is in the postcondition (θ' is an origin of θ); moreover, if $(\theta, u, \theta') \in R$ then additionally it holds that S modifies no “relevant” variable or field name, where a “relevant” variable is one occurring in the *consequent* of θ' . For example, if S is $x := w$ then R might contain the triplets $(q > 4 \Rightarrow w \times, m, q > 4 \Rightarrow x \times)$ and $(w > 3 \Rightarrow z \times, u, x > 3 \Rightarrow z \times)$.

⁴To emphasize the connection to traditional Hoare style, we use curly brackets around the pre- and postconditions; recall that those denote sets of 2-assertions.

Verification conditions (VCs). These are either of the form $\phi \triangleright^1 \phi'$, meaning that ϕ logically implies ϕ' , or of the form $\Theta \triangleright^2 \theta$, again meaning that Θ logically implies θ but now for 2-assertions. Thus $\models \phi \triangleright^1 \phi'$ iff for all s, r : whenever $s, r \models \phi$ then also $s, r \models \phi'$; and $\models \Theta \triangleright^2 \theta$ iff for all s, r, s_1, r_1, β : whenever $s, r \& s_1, r_1 \models_{\beta} \Theta$ then also $s, r \& s_1, r_1 \models_{\beta} \theta$. We use VC to range over sets of verification conditions, and write $\models VC$ iff $\models vc$ holds for all $vc \in VC$.

Now assume that some vc in the output of VCgen cannot be satisfied. (This is the only way that VCgen can “fail” on a well-typed program.) Looking at the clauses, we see that vc must have been generated by either **open** or **while**. The former case would reflect the failure to prove that \mathcal{I} is indeed a flow invariant for objects in the heap; the user would then need to propose another object flow invariant. The latter case would reflect the failure to prove that the given postcondition is indeed a loop flow invariant; the user would then need to strengthen it. The above situations are the only places where VCgen needs user assistance.

4.1 Correctness Results

Ultimately, we must express that if $[VC]\{\Theta\} (-) \Leftarrow S \{\Theta'\}$ with $\models VC$ then Θ is indeed a precondition that is strong enough to establish Θ' . (Θ may not be the *weakest* such precondition, however.) For record commands, this is stated as:

Proposition 4.1 (Correctness of record commands) *Assume that*

1. $[VC]\{\Theta\} (-) \Leftarrow RS \{\Theta'\}$ and that $\models VC$
2. $s, r \llbracket RS \rrbracket s', r'$ and $s_1, r_1 \llbracket RS \rrbracket s'_1, r'_1$
3. $s, r \& s_1, r_1 \models_{\beta} \Theta$.

Then $s', r' \& s'_1, r'_1 \models_{\beta} \Theta'$.

Note that Proposition 4.1 is termination-*insensitive*, as is also Theorem 4.2; this is not surprising given our choice of a relational semantics (but see [3] for a logic-based approach that is termination-sensitive).

Proposition 4.1 is used to prove correctness of top-level commands, for which the correctness statement is slightly more complex:

Theorem 4.2 (Correctness) *Assume that*

1. $[VC]\{\Theta\} (-) \Leftarrow TS \{\Theta'\}$ and that $\models VC$
2. $s, h \llbracket TS \rrbracket s', h'$ and that $s_1, h_1 \llbracket TS \rrbracket s'_1, h'_1$
3. $s \& s_1 \models_{\beta} \Theta$ and $h \& h_1 \models_{\beta} \mathcal{I}$.
4. There exists $\theta'_0 \in \Theta'$ such that $s' \models \text{ant}(\theta'_0)$ and $s'_1 \models \text{ant}(\theta'_0)$.

Then there exists β' extending β such that $s' \& s'_1 \models_{\beta'} \Theta'$ and $h' \& h'_1 \models_{\beta'} \mathcal{I}$.

If TS contains no **new** commands, we may choose $\beta' = \beta$, but otherwise β' may be a proper extension of β so as to model that new heap locations have been allocated. Condition 4 is a bit nonintuitive, but it is (at least currently) needed for the proofs to carry through, and it is non-restrictive as it can be fulfilled by adding to Θ' a trivial 2-assertion $\text{true} \Rightarrow 0 \times$.

Theorem 4.2 is proved in Appendix A, by establishing a number of auxiliary properties. These properties have largely determined the design of VCgen and will thus guide us as we later explain the various clauses of Figs. 5 & 6.

The first such property is a variant of the “*-property” by Bell and La Padula [9], also called “write confinement” [4], which is used to preclude, e.g., “low writes under high guards”. In our setting, it captures one role of the R component and reads as follows:

Lemma 4.3 (Totality and Write Confinement)

Assume $[VC]\{\Theta\} (R) \Leftarrow S \{\Theta'\}$. Then $\text{dom}(R) = \Theta$ and $\text{ran}(R) = \Theta'$. Given $\theta' \in \Theta'$, there exists at most one θ such that $(\theta, u, \theta') \in R$. If there exists such θ , then $\text{con}(\theta) = \text{con}(\theta')$, and with $E = \text{con}(\theta)$ we have

- if $s, - \llbracket S \rrbracket s', -$ then s agrees with s' on $\text{fv}(E)$;
- if $s, r \llbracket S \rrbracket s', r'$ (thus S is of form RS) then also r agrees with r' on $\text{ff}(E)$.

Lemma 4.3 is needed in the proof of Theorem 4.2 (and Prop. 4.1) to handle the case where the two runs in question follow *different branches* in a conditional, as we must then ensure that neither run modifies a variable (field name) on which we want the two runs to agree afterwards.

We now embark on explaining the various clauses of VCgen in Figs. 5 and 6. For an assignment $x := A$, each 2-assertion $\phi \Rightarrow E \times$ in Θ' produces exactly one 2-assertion in Θ , given by substituting A for x (as in standard Hoare logic) in ϕ as well as in E ; the connection is tagged m when x occurs in E . The treatment of field update is similar, and of **skip** even simpler. The rule for $S_1 ; S_2$ works backwards, first computing the precondition for S_2 which is then used to compute the precondition for S_1 ; the tags express that a consequent is modified iff it has been modified in either S_1 or S_2 . The rule for **assert** allows us to weaken 2-assertions, by strengthening their antecedents; this is sound since execution will abort from states not satisfying the new antecedents.

To motivate the treatment (Fig. 5) of a conditional **if** B **then** S_1 **else** S_2 , assume that $\phi \Rightarrow E \times$ occurs in Θ' . First we partition Θ' into two sets, Θ'_m and Θ'_u ; a 2-assertion can be in the latter set if its consequent is not modified by the conditional. If $(\phi \Rightarrow E \times) \in \Theta'_u$, we can assume from Lemma 4.3 that the precondition of each S_i will contain a 2-assertion of the form $\phi_i \Rightarrow E \times$; these can now be combined by R_0 into one single precondition. On the other hand, if $(\phi \Rightarrow E \times) \in \Theta'_m$ then E has been modified by at least one branch; therefore, we should not allow two runs to take *different branches* if they both satisfy ϕ afterwards. This is ensured by R'_0 , while R'_1 (R'_2) caters for the case where both runs choose S_1 (S_2).

Example 4.4 Consider the result of applying VCgen to the body of the **while** loop in Fig 1(a), with post-condition $\{x \times, \text{odd}(i) \Rightarrow v \times\}$. Working backwards, the assignment to i transforms $\text{odd}(i) \Rightarrow v \times$ to $\text{odd}(i + 1) \Rightarrow v \times$, which amounts to $\neg \text{odd}(i) \Rightarrow v \times$, but keeps $x \times$ unchanged. To process the conditional, we apply VCgen to the branches; the **else** branch produces R_2 containing

$$\begin{aligned} &(x \times, u, x \times) \\ &(\neg \text{odd}(i) \Rightarrow x \times, m, \neg \text{odd}(i) \Rightarrow v \times) \end{aligned}$$

while the **then** branch produces R_1 containing

$$\begin{aligned} &(x \times, u, x \times) \\ &(\neg \text{odd}(i) \Rightarrow (v + h) \times, m, \neg \text{odd}(i) \Rightarrow v \times) \end{aligned}$$

Referring to the clause for **if** in Fig. 5, we have $\Theta'_u = \{x \times\}$ and $\Theta'_m = \{\neg \text{odd}(i) \Rightarrow v \times\}$. The former contributes, by R_0 , the precondition $(\text{odd}(i) \vee \neg \text{odd}(i)) \Rightarrow x \times$ which amounts to $x \times$. The latter contributes by R'_1 the precondition $(\neg \text{odd}(i) \wedge \text{odd}(i)) \Rightarrow (v + h) \times$ which is vacuously true, by R'_2 the precondition $(\neg \text{odd}(i) \wedge \neg \text{odd}(i)) \Rightarrow x \times$ which amounts to $\neg \text{odd}(i) \Rightarrow x \times$, and by R'_0 the precondition $((\neg \text{odd}(i) \wedge \text{odd}(i)) \vee (\neg \text{odd}(i) \wedge \neg \text{odd}(i))) \Rightarrow \text{odd}(i) \times$ which is always true (as two states satisfying $\neg \text{odd}(i)$ will agree on the value of $\text{odd}(i)$). Assuming VCgen is able to carry out such basic simplifications, it will return, for the body of the **while** loop, an R component containing

$$\begin{aligned} &(x \times, u, x \times) \\ &(\neg \text{odd}(i) \Rightarrow x \times, m, \text{odd}(i) \Rightarrow v \times) \end{aligned}$$

The noteworthy part is that even though the postcondition mentions $v \times$, and v is updated using h , VCgen generates a precondition which does not mention h , since it exploits the parity of i .

Example 4.5 Continuing Example 3.2, it is easy to see that VCgen, when applied to the command

if $pin = 1234$ **then** $out := x$ **else** $out := y$

and the postcondition $pin = 1234 \Rightarrow out \times$, returns the precondition

$$\begin{aligned} &\{pin = 1234 \wedge pin = 1234 \Rightarrow x \times, \\ &pin = 1234 \wedge pin \neq 1234 \Rightarrow y \times, \\ &(pin = 1234 \wedge pin = 1234) \vee (pin = 1234 \wedge pin \neq 1234) \Rightarrow (pin = 1234) \times\} \end{aligned}$$

where the second assertion is vacuously true, and the third assertion is trivially true. Similarly, given the postcondition $pin \neq 1234 \Rightarrow out \times$, VCgen returns the precondition

$$\begin{aligned} &\{pin \neq 1234 \wedge pin = 1234 \Rightarrow x \times, \\ &pin \neq 1234 \wedge pin \neq 1234 \Rightarrow y \times, \\ &(pin \neq 1234 \wedge pin = 1234) \vee (pin \neq 1234 \wedge pin \neq 1234) \Rightarrow (pin = 1234) \times\} \end{aligned}$$

where the first assertion is vacuously true, and the third assertion is trivially true (as two runs which agree that $pin \neq 1234$ also agree on the value of the test $pin = 1234$). In short, assuming that VCgen can carry out basic simplifications, it will transform the postcondition $\{pin = 1234 \Rightarrow out \times, pin \neq 1234 \Rightarrow out \times\}$ into the precondition $\{pin = 1234 \Rightarrow x \times, pin \neq 1234 \Rightarrow y \times\}$.

For a **while** loop (Fig. 6), VCgen checks whether the given postcondition Θ can indeed serve as a flow invariant. (As mentioned earlier this may fail in which case the user must strengthen the postcondition.) First we partition Θ into two sets, Θ_m and Θ_u ; a 2-assertion can be in the latter set if its consequent is not modified by the loop body. Now VC_2 serves a similar function as R'_0 did in the clause for conditionals: by demanding a precondition with the loop test B as consequent, it ensures that if one run stays in the loop and updates a variable on which the two runs must agree, then also the other run stays in the loop. When both runs stay in the loop, VC_1 ensures that the loop flow invariant is maintained.

The need for VC_3 , VC_4 and VC_5 is less obvious, but they are designed so as to establish an auxiliary result, stated below as Lemma 4.6. VC_3 demands that Θ_m contains an assertion θ_m with a “weakest” antecedent. (This is no serious restriction, since if $\Theta_m = \{\phi_i \Rightarrow E_i \times \mid i \in \{1 \dots n\}\}$ we can just add $(\phi_1 \vee \dots \vee \phi_n) \Rightarrow 0 \times$ to Θ_m .)

Lemma 4.6 Assume $[VC]\{\Theta\} (R) \Leftarrow S \{\Theta'\}$ with $\models VC$. Given $\theta' \in \Theta'$, there exists $(\theta, \neg, \theta') \in R$ such that

- if $S = RS$: whenever $s, r \llbracket S \rrbracket s', r'$ and $s', r' \models \text{ant}(\theta')$ then $s, r \models \text{ant}(\theta)$;
- if $S = TS$: whenever $s, h \llbracket S \rrbracket s', h'$ and $s' \models \text{ant}(\theta')$ then $s \models \text{ant}(\theta)$.

For $S = \mathbf{while} B \mathbf{do} S_0$, if $\theta' \in \Theta_u$ we can use $\theta = \theta'$, otherwise we can use $\theta = \theta_m$.

We now address the clause for **open** x **in** RS **close**, where we first compute in Θ_0 a precondition for RS , given a postcondition that is augmented with \mathcal{I} (as the object flow invariant must be re-established at the end). Note that we must remove from Θ_0 any references to field names; for that purpose we assume that there is a function $ff^+ : \mathbf{1Assert} \rightarrow \mathbf{1Assert}$ such that if $\phi' = ff^+(\phi)$ then (i) ϕ' is field-free, and (ii) ϕ logically implies ϕ' . These demands are trivially fulfilled if $ff^+(\phi) = \text{true}$ for all ϕ , but a more precise solution is possible; then, e.g., ff^+ returns $x = 7$ given $x = 7 \wedge \neg(.f = 8)$. Thus, e.g., Θ will (by R_3 or R_4) contain $x = 7 \Rightarrow y \times$ if Θ_0 contains $(x = 7 \wedge \neg(.f = 8)) \Rightarrow y \times$.

Equipped with ff^+ , we can explain the various clauses, first R_4 which “lifts out” assertions in Θ_0 that originate from a top-level assertion and whose consequents have not been modified. Now consider an assertion in Θ_0 whose consequent *has* been modified. If the resulting consequent is not field-free, we must demand that it follows from the object flow invariant, as expressed by VC_2 . Otherwise, it can be lifted out of the scope, as done by R_3 . A precondition, say $(.f + x) \times$ might need to be replaced by the two assertions $x \times$ and $.f \times$ which together are strictly stronger; the former can be lifted out, the latter must follow from \mathcal{I} . Also, assertions in \mathcal{I} whose consequents have not been modified (and therefore still contain field names) must follow from \mathcal{I} , as expressed by VC_1 . The role of R_1 and R_2 is to ensure that if a relevant variable (in Θ' or in \mathcal{I}) is modified, the two runs are indeed manipulating the same object.

Note that R_2 ensures that there are “ m ” tags going out from *all* 2-assertions in the postcondition of a command that modifies a consequent of a 2-assertion in \mathcal{I} . This property is required by the following Lemma:

Lemma 4.7 *Assume $[VC]\{\Theta\} (R) \Leftarrow TS \{\Theta'\}$ with $\models VC$, and that there exists $\theta' \in \Theta'$ such that if $(-, \gamma, \theta') \in R$ then $\gamma = u$. For $(\phi_0 \Rightarrow E_0 \times) \in \mathcal{I}$, if $s, h \llbracket TS \rrbracket s', h'$ then for all $\ell \in \text{dom}(h)$:*

- if $h'(\ell) \models \phi_0$ then $h(\ell) \models \phi_0$;
- $h(\ell)(f) = h'(\ell)(f)$ for all f in $\text{ff}(E_0)$.

To see why Lemma 4.7 is needed, recall that the correctness of **if** and **while** rests on Lemma 4.3 which ensures that if two runs follow different paths then they do not modify consequents of top-level assertions. Lemma 4.7 now further ensures that two such diverting runs do not invalidate object flow invariants.

The clause for **new** x **in** RS **close** first computes in Θ_0 a precondition for RS , and then exploits that the semantics of **new** initializes all fields to a default value. So if Θ_0 contains say $.f = 1 \Rightarrow y \times$, we generate the (vacuously true) precondition $0 = 1 \Rightarrow y \times$; if Θ_0 contains say $(.f + y) \times$, we generate the precondition $(0 + y) \times$. We also want to eliminate x from the precondition; this is possible due to the freshness of the new location and the absence of pointer arithmetic: after object allocation, $x = A$ can hold only if A is x . For the antecedents, this is formalized by the function $rm_x^\ell : \mathbf{1Assert} \rightarrow \mathbf{1Assert}$ which is a homomorphism on the structure of ϕ , which maps $(x = x)$ to *true*, which maps $(x = A)$ and $(A = x)$ to *false* if A is not x and hence $x \notin \text{fv}(A)$, and which maps any B not containing x to itself. For the consequents, we also exploit that two runs will always agree on the value of x after allocation (as β can be extended to relate the fresh locations), implying that the two runs will agree on the value of any expression containing x . This is formalized by the function $rm_x^r : \mathbf{Exp} \rightarrow \mathbf{Exp}$ which maps E into 0 if $x \in \text{fv}(E)$, and into E otherwise.

4.2 Strengthening and Simplifying Assertions

As can be seen by inspecting, e.g., the clause for conditionals, the preconditions generated by VCgen may contain a number of assertions which is exponential in the size of the program. Our implementation therefore needs to be able to simplify assertions, replacing a precondition with one which is equivalent. In particular, it is important (cf. Examples 4.4 and 4.5) to recognize when a 2-assertion has an antecedent which is always false, or when it is of the form $\phi \Rightarrow B \times$ where ϕ implies B (or $\neg B$), since then it can be eliminated. Preliminary experiments with our prototype implementation indicate that a few such rules are sufficient to yield readable preconditions; this makes us hope for a running time which is close to linear though further experiments are needed.

Let us be more formal about what must hold when θ is replaced by $\theta_1 \dots \theta_n$, where we record in R that θ is related to each θ_i . Conceptually, we can view such a replacement as the result of VCgen processing an invisible “**simplify**” command with the same semantics as “**skip**”; thus we must ensure that all our technical results hold also when $S = \mathbf{simplify}$ (which is trivial for Lemma 4.7). For Proposition 4.1 and Theorem 4.2, this can be achieved by demanding that $\{\theta_1, \dots, \theta_n\} \triangleright^2 \theta$. For Lemma 4.3, we must require that if some θ_i is assigned the tag u then $i = 1$ and $con(\theta) = con(\theta_1)$, and also require that $n \geq 1$. For Lemma 4.6, we must require that for at least one $i \in \{1 \dots n\}$ we can verify $ant(\theta) \triangleright^1 ant(\theta_i)$. As a consequence of these demands, rather than eliminating a 2-assertion which is always true, we replace it by a designated such assertion, e.g., $true \Rightarrow 0 \times$. Also note that to simplify $\phi \Rightarrow (x + w) \times$ into $\phi \Rightarrow x \times$ and $\phi \Rightarrow w \times$, the connections must be tagged m which is somewhat counterintuitive as neither x nor w is modified; it appears possible to amend our theory so as to allow such connections to be tagged u , but the price will be a more complex statement of Lemma 4.3 (also the definition of R_0 in the clause for conditionals in Fig. 5 would need to be changed).

5 Worked Out Example

In this section we work out the examples given in Sect. 2.

5.1 Analyzing Fig. 1(b)

We want to prove that the program satisfies the specification $\{x \times\} - \{odd(i) \Rightarrow res \times\}$. The object flow invariant, \mathcal{I} , is a conjunction of $odd(.idx) \Rightarrow .val \times$ and $odd(.idx) \Rightarrow .src \times$.

We first consider the last **open**, lines 9–13 of Fig. 1(b), where we must analyze the body (lines 10–12) with a postcondition which is $odd(i) \Rightarrow res \times$ conjoined with the object flow invariant. Using VCgen’s clauses for assignment, field update, and **assert**, this yields an empty set of VCs, and R_0 containing

$$\begin{aligned} & (odd(i) \wedge (.idx = i) \Rightarrow q \times, \quad m, \quad odd(i) \Rightarrow res \times) \\ & (odd(.idx) \wedge (.idx = i) \Rightarrow q \times, \quad m, \quad odd(.idx) \Rightarrow .val \times) \\ & (odd(.idx) \wedge (.idx = i) \Rightarrow .src \times, \quad u, \quad odd(.idx) \Rightarrow .src \times) \end{aligned}$$

Applying the clause in VCgen for **open** now generates the valid VC

$$VC_1 = \{odd(.idx) \wedge (.idx = i) \triangleright^1 odd(.idx)\}$$

whereas VC_2 is empty (because the relevant assertions are of the form $- \Rightarrow q \times$ but $q \times$ is field-free). Also, it generates a set R which is the union of the sets R_1, R_2, R_3 below (since R_4 is empty).

$[VC]\{\Theta\} (R) \Leftarrow \mathbf{skip} \{\Theta'\}$
 iff $R = \{(\theta, u, \theta) \mid \theta \in \Theta'\}$ and $\Theta = \Theta'$ and $VC = \emptyset$

$[VC]\{\Theta\} (R) \Leftarrow \mathbf{assert}(\phi_0) \{\Theta'\}$
 iff $R = \{((\phi \wedge \phi_0) \Rightarrow E \times, u, \phi \Rightarrow E \times) \mid (\phi \Rightarrow E \times) \in \Theta'\}$
 and $\Theta = \text{dom}(R)$ and $VC = \emptyset$

$[VC]\{\Theta\} (R) \Leftarrow x := A \{\Theta'\}$
 iff $R = \{(\phi[A/x] \Rightarrow E[A/x] \times, \gamma, \phi \Rightarrow E \times) \mid (\phi \Rightarrow E \times) \in \Theta'\}$
 where $\gamma = m$ iff $x \in \text{fv}(E)$
 and $\Theta = \text{dom}(R)$ and $VC = \emptyset$

$[VC]\{\Theta\} (R) \Leftarrow .f := A \{\Theta'\}$
 iff $R = \{(\phi[A/.f] \Rightarrow E[A/.f] \times, \gamma, \phi \Rightarrow E \times) \mid (\phi \Rightarrow E \times) \in \Theta'\}$
 where $\gamma = m$ iff $f \in \text{ff}(E)$
 and $\Theta = \text{dom}(R)$ and $VC = \emptyset$

$[VC]\{\Theta\} (R) \Leftarrow S_1 ; S_2 \{\Theta'\}$
 iff $[VC_2]\{\Theta''\} (R_2) \Leftarrow S_2 \{\Theta'\}$ and $[VC_1]\{\Theta\} (R_1) \Leftarrow S_1 \{\Theta'\}$
 and $R = \{(\theta, \gamma, \theta') \mid \exists \theta'', \gamma_1, \gamma_2 : (\theta, \gamma_1, \theta'') \in R_1, (\theta'', \gamma_2, \theta') \in R_2\}$
 where $\gamma = m$ iff $\gamma_1 = m$ or $\gamma_2 = m$
 and $VC = VC_1 \cup VC_2$

$[VC]\{\Theta\} (R) \Leftarrow \mathbf{if} B \mathbf{then} S_1 \mathbf{else} S_2 \{\Theta'\}$
 iff $[VC_1]\{\Theta_1\} (R_1) \Leftarrow S_1 \{\Theta'\}$ and $[VC_2]\{\Theta_2\} (R_2) \Leftarrow S_2 \{\Theta'\}$
 and $R = R'_1 \cup R'_2 \cup R'_0 \cup R_0$
 where $R'_1 = \{((\phi_1 \wedge B) \Rightarrow E_1 \times, m, \theta') \mid \theta' \in \Theta'_m, (\phi_1 \Rightarrow E_1 \times, -, \theta') \in R_1\}$
 and $R'_2 = \{((\phi_2 \wedge \neg B) \Rightarrow E_2 \times, m, \theta') \mid \theta' \in \Theta'_m, (\phi_2 \Rightarrow E_2 \times, -, \theta') \in R_2\}$
 and $R'_0 = \{(((\phi_1 \wedge B) \vee (\phi_2 \wedge \neg B)) \Rightarrow B \times, m, \theta') \mid \theta' \in \Theta'_m, (\phi_1 \Rightarrow E_1 \times, -, \theta') \in R_1, (\phi_2 \Rightarrow E_2 \times, -, \theta') \in R_2\}$
 and $R_0 = \{(((\phi_1 \wedge B) \vee (\phi_2 \wedge \neg B)) \Rightarrow E \times, u, \theta') \mid \theta' \in \Theta'_u, (\phi_1 \Rightarrow E \times, u, \theta') \in R_1, (\phi_2 \Rightarrow E \times, u, \theta') \in R_2\}$
 and $\Theta'_m = \{\theta' \in \Theta' \mid \exists (-, m, \theta') \in R_1 \cup R_2\}$
 and $\Theta'_u = \Theta' \setminus \Theta'_m$
 and $\Theta = \text{dom}(R)$ and $VC = VC_1 \cup VC_2$

Figure 5: The verification condition generator, part I

$$\begin{aligned}
 R_1 &= \{(odd(i) \Rightarrow x \times, \quad m, \quad odd(i) \Rightarrow res \times)\} \\
 R_2 &= \{(true \Rightarrow x \times, \quad m, \quad odd(i) \Rightarrow res \times)\} \\
 R_3 &= \{(odd(i) \Rightarrow q \times, \quad m, \quad odd(i) \Rightarrow res \times)\}
 \end{aligned}$$

We have assumed that ff^+ maps $odd(.idx) \wedge (.idx = i)$, and $odd(i) \wedge (.idx = i)$, into $odd(i)$. Now the precondition of lines 9–13 can be read off from the above sets as $\{odd(i) \Rightarrow x \times, x \times, odd(i) \Rightarrow q \times\}$ where the first assertion can be removed as it follows from the second, leaving us with $\{x \times, odd(i) \Rightarrow q \times\}$. Next, we analyze lines 5–8 of Fig. 1(b) with the above as postcondition. For lines 6–7, apart from the precondition and an empty set of VCs, VCgen generates R_0 containing

$[VC]\{\Theta\} (R) \Leftarrow \mathbf{while} B \mathbf{do} S_0 \{\Theta\}$
iff $[VC_0]\{\Theta_0\} (R_0) \Leftarrow S_0 \{\Theta\}$
and $R = \{(\theta, u, \theta) \mid \theta \in \Theta_u\} \cup \{(\theta_1, m, \theta_2) \mid \theta_1, \theta_2 \in \Theta_m\}$
and $VC = VC_0 \cup VC_1 \cup VC_2 \cup VC_3 \cup VC_4 \cup VC_5$
where $VC_1 = \{\Theta \triangleright^2 (\phi \wedge B) \Rightarrow E \times \mid (\phi \Rightarrow E \times, -, -) \in R_0\}$
and $VC_2 = \{\Theta \triangleright^2 \phi_m \Rightarrow B \times\}$
and $VC_3 = \{ant(\theta) \triangleright^1 \phi_m \mid \theta \in \Theta_m\}$
and $VC_4 = \{ant(\theta) \triangleright^1 \phi_m \mid (\theta, -, \theta_m) \in R_0\}$
and $VC_5 = \{ant(\theta_0) \triangleright^1 ant(\theta) \mid (\theta_0, -, \theta) \in R_0, \theta \in \Theta_u\}$
and $\Theta_m = \{\theta \in \Theta \mid \exists(-, m, \theta) \in R_0\}$
and $\Theta_u = \Theta \setminus \Theta_m$
and Θ_m contains a special element θ_m with $\phi_m = ant(\theta_m)$

$[VC]\{\Theta\} (R) \Leftarrow \mathbf{open} x \mathbf{in} RS \mathbf{close} \{\Theta'\}$
iff $[VC_0]\{\Theta_0\} (R_0) \Leftarrow RS \{\Theta' \cup \mathcal{I}\}$
and $R = R_1 \cup R_2 \cup R_3 \cup R_4$
where $R_1 = \{\{ff^+(\phi) \Rightarrow x \times, m, \theta'\} \mid \theta' \in \Theta', (\phi \Rightarrow -, m, \theta') \in R_0\}$
and $R_2 = \{\text{if exists } \theta \in \mathcal{I} \text{ with } (-, m, \theta) \in R_0 \text{ then } \{\{true \Rightarrow x \times, m, \theta'\} \mid \theta' \in \Theta'\} \text{ else } \emptyset\}$
and $R_3 = \{\{ff^+(\phi) \Rightarrow E \times, m, \theta'\} \mid \theta' \in \Theta', E \text{ field-free, } \exists \theta'_0 \in \mathcal{I} \cup \{\theta'\} : (\phi \Rightarrow E \times, m, \theta'_0) \in R_0\}$
and $R_4 = \{\{ff^+(\phi) \Rightarrow E \times, u, \theta'\} \mid \theta' \in \Theta', (\phi \Rightarrow E \times, u, \theta') \in R_0\}$
and $\Theta = dom(R)$ and $VC = VC_0 \cup VC_1 \cup VC_2$
where $VC_1 = \{ant(\theta) \triangleright^1 ant(\theta') \mid \theta' \in \mathcal{I}, (\theta, u, \theta') \in R_0\}$
and $VC_2 = \{\mathcal{I} \triangleright^2 \theta \mid (\theta, m, -) \in R_0, con(\theta) \text{ not field-free}\}$

$[VC]\{\Theta\} (R) \Leftarrow \mathbf{new} x \mathbf{in} RS \mathbf{close} \{\Theta'\}$
iff $[VC_0]\{\Theta_0\} (R_0) \Leftarrow RS \{\Theta' \cup \mathcal{I}\}$
and $R = \{(rm_x^l(\phi[\overline{deflt}(f)/\bar{f}]) \Rightarrow rm_x^r(E[\overline{deflt}(f)/\bar{f}]) \times, \gamma, \theta') \mid (\phi \Rightarrow E \times, \gamma_0, \theta') \in R_0, \gamma = m \text{ iff } \gamma_0 = m \text{ or } x \in fv(E)\}$
and $\Theta = dom(R)$ and $VC = VC_0$

Figure 6: The verification condition generator, part II

$$\begin{aligned}
& (odd(i) \rightarrow odd(.idx) \Rightarrow x \times, \quad u, \quad true \Rightarrow x \times) \\
& (odd(i) \wedge (odd(i) \rightarrow odd(.idx)) \Rightarrow .val \times, \quad m, \quad odd(i) \Rightarrow q \times) \\
& (odd(.idx) \wedge (odd(i) \rightarrow odd(.idx)) \Rightarrow .val \times, \quad u, \quad odd(.idx) \Rightarrow .val \times) \\
& (odd(.idx) \wedge (odd(i) \rightarrow odd(.idx)) \Rightarrow .src \times, \quad u, \quad odd(.idx) \Rightarrow .src \times)
\end{aligned}$$

Applying the clause in VCgen for **open** now generates the valid VCs

$$\begin{aligned}
VC_1 &= \{odd(.idx) \wedge (odd(i) \rightarrow odd(.idx)) \triangleright^1 odd(.idx)\} \\
VC_2 &= \{\mathcal{I} \triangleright^2 (odd(i) \wedge (odd(i) \rightarrow odd(.idx))) \Rightarrow .val \times\}
\end{aligned}$$

and a set R which is the union of R_1 and R_4 below (since R_2 and R_3 are empty).

$$\begin{aligned}
R_1 &= \{(odd(i) \Rightarrow y \times, \quad m, \quad odd(i) \Rightarrow q \times)\} \\
R_4 &= \{(true \Rightarrow x \times, \quad u, \quad true \Rightarrow x \times)\}
\end{aligned}$$

We have assumed that ff^+ maps $odd(i) \rightarrow odd(.idx)$ into $true$, and maps $odd(i) \wedge (odd(i) \rightarrow$

$odd(.idx)$ into $odd(i)$. Now the precondition of lines 5–8 can be read off as $\{odd(i) \Rightarrow y \times, x \times\}$. Finally, we analyze lines 1–4 of Fig. 1(b) with the above as postcondition. For lines 2–3, VCgen generates R_0 containing

$$\begin{aligned} & (odd(.idx) \Rightarrow .src \times, \quad m, \quad odd(i) \Rightarrow y \times) \\ & \quad (true \Rightarrow x \times, \quad u, \quad true \Rightarrow x \times) \\ & (odd(.idx) \Rightarrow .src \times, \quad u, \quad odd(.idx) \Rightarrow .src \times) \\ & (odd(.idx) \Rightarrow .val \times, \quad u, \quad odd(.idx) \Rightarrow .val \times) \end{aligned}$$

Applying the clause in VCgen for **open** now generates the valid VCs

$$\begin{aligned} VC_1 &= \{odd(.idx) \triangleright^1 odd(.idx)\} \\ VC_2 &= \{\mathcal{I} \triangleright^2 odd(.idx) \Rightarrow .src \times\} \end{aligned}$$

and a set R which is the union of R_1 and R_4 below (since R_2 and R_3 are empty).

$$\begin{aligned} R_1 &= \{(true \Rightarrow x \times, \quad m, \quad odd(i) \Rightarrow y \times)\} \\ R_4 &= \{(true \Rightarrow x \times, \quad u, \quad true \Rightarrow x \times)\} \end{aligned}$$

Now the overall precondition can be read off as $x \times$, and we note that all the VCs generated are valid. Fig. 7 shows the assertions that hold at each line in the program.

5.2 Analyzing Fig. 1(a)

Recall that the program in question is

```

i := 0; v := 0; res := 0;
while (i < 7) do
  if odd(i)
  then
    res := res + v;
    v := v + h;
  else
    v := x;
  fi;
  i := i + 1;
od

```

It turns out that we can use the loop flow invariant Θ given by

$$\Theta = \{odd(i) \Rightarrow v \times, i \times, res \times, x \times\}$$

which implies the desired postcondition, $res \times$. To apply VCgen to the while loop, we must apply it recursively to the loop body; this involves applying VCgen to the conditional where the postcondition is now (due to i being incremented) given by

$$\{odd(i + 1) \Rightarrow v \times, (i + 1) \times, res \times, x \times\}.$$

For that purpose, we first analyze the **then** branch, yielding R_1 containing

$$\begin{aligned} & (odd(i + 1) \Rightarrow (v + h) \times, \quad m, \quad odd(i + 1) \Rightarrow v \times) \\ & \quad ((i + 1) \times, \quad u, \quad (i + 1) \times) \\ & ((res + v) \times, \quad m, \quad res \times) \\ & \quad (x \times, \quad u, \quad x \times) \end{aligned}$$

```

    {true ⇒ x×}
1.  open x in
    {odd(.idx) ⇒ .src×, true ⇒ x×, odd(.idx) ⇒ .val×}
2.      y := .src;
    //Case of field access: replace y by .src to obtain pre
    {odd(.idx) ⇒ y×, true ⇒ x×, odd(.idx) ⇒ .val×, odd(.idx) ⇒ .src×}
3.      i := .idx;
    //Case of field access: replace i by .idx to obtain pre
    {odd(i) ⇒ y×, true ⇒ x×, odd(.idx) ⇒ .val×, odd(.idx) ⇒ .src×}
    // Conjoin object flow invariant to post
4.  close;
    {odd(i) ⇒ y×, true ⇒ x×}
5.  open y in
    {(odd(i) → odd(.idx)) ⇒ x×,
     odd(i) ∧ (odd(i) → odd(.idx)) ⇒ .val×,
     odd(.idx) ∧ (odd(i) → odd(.idx)) ⇒ .val×,
     odd(.idx) ∧ (odd(i) → odd(.idx)) ⇒ .src×}
6.      assert (odd(i) → odd(.idx));
    //Conjoin assertion to obtain pre
    {true ⇒ x×, odd(i) ⇒ .val×, odd(.idx) ⇒ .val×, odd(.idx) ⇒ .src×}
7.      q := .val;
    //Case of field access: replace q by .val to obtain pre
    {true ⇒ x×, odd(i) ⇒ q×, odd(.idx) ⇒ .val×, odd(.idx) ⇒ .src×}
    // Conjoin object flow invariant to simplified post
8.  close;
    {odd(i) ⇒ x×, true ⇒ x×, odd(i) ⇒ q×}
9.  open x in
    {odd(i) ∧ (.idx = i) ⇒ q×, odd(.idx) ∧ (.idx = i) ⇒ q×,
     odd(.idx) ∧ (.idx = i) ⇒ .src×}
10.     assert (.idx = i);
    //Conjoin assertion to obtain pre
    {odd(i) ⇒ q×, odd(.idx) ⇒ q×, odd(.idx) ⇒ .src×}
11.     .val := q;
    //Case of field update: replace .val by q to obtain pre
    {odd(i) ⇒ .val×, odd(.idx) ⇒ .val×, odd(.idx) ⇒ .src×}
12.     res := .val;
    //Case of field access: replace res by .val to obtain pre
    {odd(i) ⇒ res×, odd(.idx) ⇒ .val×, odd(.idx) ⇒ .src×}
    // Conjoin object flow invariant to post
13.  close;
    {odd(i) ⇒ res×}

```

Figure 7: Applying VCgen to Fig. 1(b).

and next analyze the else-branch, yielding R_2 containing

$$\begin{aligned} & (odd(i+1) \Rightarrow x \times, \quad m, \quad odd(i+1) \Rightarrow v \times) \\ & \quad ((i+1) \times, \quad u, \quad (i+1) \times) \\ & \quad \quad (res \times, \quad u, \quad res \times) \\ & \quad \quad \quad (x \times, \quad u, \quad x \times) \end{aligned}$$

The rule for conditionals now returns $R = R'_1 \cup R'_2 \cup R'_0 \cup R_0$ where

$$\begin{aligned} R'_1 &= \{(odd(i+1) \wedge odd(i) \Rightarrow (v+h) \times, m, odd(i+1) \Rightarrow v \times), \\ & \quad (odd(i) \Rightarrow (res+v) \times, m, res \times)\} \\ R'_2 &= \{(odd(i+1) \wedge \neg odd(i) \Rightarrow x \times, m, odd(i+1) \Rightarrow v \times), \\ & \quad (\neg odd(i) \Rightarrow res \times, m, res \times)\} \\ R'_0 &= \{(odd(i+1) \wedge odd(i) \vee odd(i+1) \wedge \neg odd(i) \Rightarrow odd(i) \times, m, odd(i+1) \Rightarrow v \times), \\ & \quad (odd(i) \vee \neg odd(i) \Rightarrow odd(i) \times, m, res \times)\} \\ R_0 &= \{(odd(i) \vee \neg odd(i) \Rightarrow (i+1) \times, u, (i+1) \times), \\ & \quad (odd(i) \vee \neg odd(i) \Rightarrow x \times, u, x \times)\} \end{aligned}$$

Therefore, after some trivial simplifications, analyzing the loop body yields R_0 containing

$$\begin{aligned} & (odd(i+1) \wedge odd(i) \Rightarrow (v+h) \times, \quad m, \quad odd(i) \Rightarrow v \times) \\ & \quad (odd(i) \Rightarrow (res+v) \times, \quad m, \quad res \times) \\ & \quad (odd(i+1) \wedge \neg odd(i) \Rightarrow x \times, \quad m, \quad odd(i) \Rightarrow v \times) \\ & \quad \quad (\neg odd(i) \Rightarrow res \times, \quad m, \quad res \times) \\ & \quad (odd(i+1) \Rightarrow odd(i) \times, \quad m, \quad odd(i) \Rightarrow v \times) \\ & \quad \quad (true \Rightarrow odd(i) \times, \quad m, \quad res \times) \\ & \quad \quad (true \Rightarrow (i+1) \times, \quad m, \quad i \times) \\ & \quad \quad \quad (true \Rightarrow x \times, \quad u, \quad x \times) \end{aligned}$$

In the rule for **while**, we have $\Theta'_u = \{x \times\}$ and $\Theta'_m = \{odd(i) \Rightarrow v \times, i \times, res \times\}$, and may use $\theta_m = i \times$ and $\phi_m = true$. Now VCgen generates several verification conditions, where the ones in VC_3 , VC_4 , and VC_5 are all of the form $_ \triangleright^1 true$, and thus trivially valid. The only VC in VC_2 is $\Theta \triangleright^2 true \Rightarrow (i < 7) \times$ which is valid since Θ implies $i \times$. The interesting VCs are those in VC_1 which require that Θ logically implies all of the below:

1. $odd(i+1) \wedge odd(i) \wedge i < 7 \Rightarrow (v+h) \times$
2. $odd(i) \wedge i < 7 \Rightarrow (res+v) \times$
3. $odd(i+1) \wedge \neg odd(i) \wedge i < 7 \Rightarrow x \times$
4. $\neg odd(i) \wedge i < 7 \Rightarrow res \times$
5. $odd(i+1) \wedge i < 7 \Rightarrow odd(i) \times$
6. $i < 7 \Rightarrow odd(i) \times$
7. $i < 7 \Rightarrow (i+1) \times$
8. $i < 7 \Rightarrow x \times$

But from Θ implying $x \times$ we get 3 and 8; from Θ implying $i \times$ we get 5 and 6 and 7; from Θ implying $res \times$ we get 4 and then, from Θ implying $odd(i) \Rightarrow v \times$, also 2. We are left with 1, which holds vacuously, thanks to $odd(i+1)$ and $odd(i)$ being mutually exclusive.

Thus Θ is also the precondition for the while loop, and VCgen when applied to the whole program returns the precondition $\Theta = \{odd(0) \Rightarrow 0 \times, 0 \times, x \times\}$ which can be simplified to $\{x \times\}$. In particular, and as desired, we do *not* need to assume $h \times$.

5.3 Analyzing Fig. 2

Recall that the object flow invariant \mathcal{I} is given by $\mathcal{I} = \{\mathcal{I}_{1v}, \mathcal{I}_{1t}, \mathcal{I}_{1n}, \mathcal{I}_{2v}, \mathcal{I}_{2n}\}$ where

$$\begin{aligned} \mathcal{I}_{1v} &= (.t = 1 \wedge .val < 10) \Rightarrow .val \times \\ \mathcal{I}_{1t} &= .t = 1 \Rightarrow (.val < 10) \times & \mathcal{I}_{1n} &= .t = 1 \Rightarrow .next \times \\ \mathcal{I}_{2v} &= .t = 2 \Rightarrow .val \times & \mathcal{I}_{2n} &= .t = 2 \Rightarrow .next \times \end{aligned}$$

For the first while loop, we shall need the invariant $\{x \times, y \times\}$ whereas for the second while loop, we shall need the invariant $\{x \times, y \times, res \times\}$. Equipped with those invariants, VCgen can successfully analyze the program of Fig. 2; in Figs. 8 and 9 we list the assertions that hold at key program points, assuming that some basic simplifications have been carried out (cf. Sect. 4.2). It will be a good exercise for the reader to work out the details.

6 Discussion

A recently popular approach to information flow analysis is *self-composition*, first proposed by Barthe et al. [8] and later extended by, e.g., Terauchi and Aiken [24] and Naumann [22]. Self-composition works as follows: for a given program S , a copy S' is created with all variables renamed (primed); with the observable variables say x, y , then NI holds provided the sequential composition $S; S'$ when given precondition $x = x' \wedge y = y'$ also ensures postcondition $x = x' \wedge y = y'$.

Terauchi and Aiken [24] use self-composition to verify information flow automatically using the BLAST tool [20]. To obtain good experimental results, they introduce sound program transformations of self-composed programs; it is also often necessary to leverage the results of a standard information flow analysis, e.g., one based on security types. In a sense, our approach is dual in that noninterference properties are explicit in our analysis but we can leverage standard assertions, inserted and/or checked by general verifiers. An interesting question is whether the 2-assertions generated by VCgen could be translated into assertions that would assist the self-composition approach.

Terauchi and Aiken [24] do not address heap-manipulating programs, so the work most closely related to ours is the one by Naumann [22] whose goal is the verification of information flow using existing verifiers like Spec# [7] or ESC/Java2 [13]. Naumann's contribution is to extend the theory of self-composition to account for manipulations of heap objects. In some cases, like for while loops, it is more practical (but not necessary) for the technique to perform program transformations. For heap-manipulating programs, the two copies of the programs involve different sets of objects and therefore the correspondence between the objects ("mates" in Naumann's terminology) must be made explicit in the specification of the composed program. Our approach avoids program transformations, and our specifications do not need to specify mates: that is handled by the semantics of assertions. On the other hand, we cannot use an existing verifier like Spec# or ESC/Java2 directly; we must thus show how preconditions and VCs are actually generated.

Dufay et al. [16] use self-composition to check noninterference for data mining algorithms implemented in Java, using the Krakatoa tool, based on the Coq theorem prover and using JML [11]. The tool reads Java input files and produces specifications for Coq and a representation of the semantics of the Java program into

```

{x⊗}
  y := nil; z := nil;
{x⊗, y⊗}
  while x ≠ nil do
{x⊗, y⊗}
    open x in
{x⊗, y⊗, v⊗, (t = 1 ∧ .val < 10 ⇒ .val⊗), (t = 1 ⇒ (.val < 10)⊗), I1v, I1t, I1n}
      assert(t = 1); v := .val; n := .next;
{x⊗, y⊗, v < 10 ⇒ v⊗, (v < 10)⊗, I}
      close;
{x⊗, y⊗, v < 10 ⇒ v⊗, (v < 10)⊗}
      x := n;
{x⊗, y⊗, v < 10 ⇒ v⊗, (v < 10)⊗}
      if v < 10
      then
{x⊗, v⊗, y⊗}
        new y1 in
{x⊗, y1⊗, v⊗, y⊗}
          .val := v; .next := y;
{x⊗, y1⊗, .val⊗, .next⊗}
          .t := 2;
{x⊗, y1⊗, I}
          close;
{x⊗, y1⊗}
          y := y1;
{x⊗, y⊗}
        else
{x⊗, y⊗}
          new z1 in
{x⊗, y⊗}
            .val := v; .next := z;
{x⊗, y⊗, (false ⇒ ...⊗)}
            .t := 3;
{x⊗, y⊗, I}
            close; z := z1;
{x⊗, y⊗}
          fi;
        od;
      {x⊗, y⊗}
    od;
{x⊗, y⊗}

```

Figure 8: Applying VCgen to Fig. 2, part I.

```

    {x⊗, y⊗}
      res := nil;
    {res⊗, x⊗, y⊗}
      while y ≠ nil do
    {x⊗, y⊗}
      open y in
    {(t = 2 ⇒ .val⊗), x⊗, (t = 2 ⇒ .next⊗), I}
      assert(.t = 2); res := .val; y := .next;
    {res⊗, x⊗, y⊗, I}
      close;
    {res⊗, x⊗, y⊗}
      od
    {res⊗}

```

Figure 9: Applying VCgen to Fig. 2, part II.

the input language of *Why*⁵, an annotated, ML-like core language with references. We leave the comparison between Krakatoa’s handling of the heap and that of ours to future work. Darvas et al. [15] use the KeY tool for interactive verification of noninterference. Information flow is modeled by a dynamic logic formula, rather than by assertions as in self-composition.

Bergeretti and Carré [10] present a compositional method for inferring dependence among variables; this technique forms the basis for the Spark Ada Examiner [12] which requires that each method is annotated with `derives` annotations like

```

derives v from y, z
      & w from x, y

```

It is interesting to observe that such “channels” of information flow are captured by our R component, as when

$$[VC]\{x\otimes, y\otimes, z\otimes\} (R) \Leftarrow S \{v\otimes, w\otimes\}$$

with R containing the elements $(y\otimes, -, v\otimes)$, $(z\otimes, -, v\otimes)$, $(x\otimes, -, w\otimes)$, $(y\otimes, -, w\otimes)$. Our approach is more general in that it also captures *conditional* channels; we are part of current work investigating how to extend the Spark Ada Examiner framework to express R elements like $(i > 5 \Rightarrow y\otimes, -, j > 7 \Rightarrow v\otimes)$. Also, we hope to investigate the relationship to the path conditions presented by Hammer et al. [19].

In the near future, we plan to experiment with the prototype implementation which is currently being developed by our student Jonathan Hoag. In future work, we might try to integrate it with the Bogor tool [17] to generate and/or check standard assertions that will increase precision. To ease expressiveness, we would like to allow multiple scopes to be simultaneously open. To handle realistic applications, we must extend the framework to an interprocedural setting.

An important long-term goal is to provide for automatic information flow analysis. This requires (i) implementing a tool to check the satisfiability of VCs; and (ii) developing techniques for the automatic computation of flow (loop/object) invariants. We expect that (ii) will involve fixed point iteration, using (i)

⁵On the web at <http://why.lri.fr/>

to check for convergence, and using a form of widening[14] to ensure termination. For (i), we would need a sound axiomatization of \triangleright^2 , where a trivial rule is that $\{\phi \Rightarrow x \times, \phi \Rightarrow w \times\} \triangleright^2 (\phi \Rightarrow (x + w) \times)$ holds for all ϕ, x, w ; it remains to be seen whether one can hope for a tool which is not just sound but also complete. Relatedly, we would like to investigate whether our analysis is in some sense “optimal”, with the preconditions being “weakest”, assuming that the program is decorated with suitable assertion statements (without such assertions our analysis can be imprecise, cf. the example at the end of Sect. 1).

References

- [1] Torben Amtoft, Sruthi Bandhakavi, and Anindya Banerjee. A logic for information flow in object-oriented programs. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 91–102, 2006. Extended version available as KSU CIS-TR-2005-1.
- [2] Torben Amtoft and Anindya Banerjee. Information flow analysis in logical form. In *SAS 2004 (11th Static Analysis Symposium)*, volume 3148 of *LNCS*, pages 100–115. Springer-Verlag, 2004.
- [3] Torben Amtoft and Anindya Banerjee. A logic for information flow analysis with an application to forward slicing of simple imperative programs. *Science of Computer Programming*, 64(1):3–28, 2007.
- [4] Anindya Banerjee and David A. Naumann. Stack-based access control and secure information flow. *Journal of Functional Programming*, 15(2):131–177, 2005. Special issue on Language Based Security.
- [5] Anindya Banerjee, David A. Naumann, and Stan Rosenberg. Towards a logical account of declassification (short paper). In *the 2007 workshop on Programming languages and analysis for security (PLAS), San Diego, California*, pages 61–66, 2007.
- [6] Michael Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.
- [7] Michael Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *Revised selected papers of CASSIS 2004*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer-Verlag, 2005.
- [8] Gilles Barthe, Pedro R. D’Argenio, and Tamara Rezk. Secure information flow by self-composition. In *IEEE Computer Security Foundations Workshop (CSFW)*, pages 100–114, 2004.
- [9] D.E. Bell and L.J. LaPadula. Secure computer systems: Mathematical foundations. Technical Report MTR-2547, MITRE Corp., 1973.
- [10] Jean-Francois Bergeretti and Bernard A. Carré. Information-flow and data-flow analysis of while-programs. *ACM Transactions on Programming Languages and Systems*, 7(1):37–61, January 1985.
- [11] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *STTT*, 7(3):212–232, 2005.
- [12] Roderick Chapman and Adrian Hilton. Enforcing security and safety models with an information flow analysis tool. In *SIGAda’04, Atlanta, Georgia*, pages 39–46. ACM, November 2004.

- [13] David R. Cok and Joseph Kiniry. ESC/Java2: Uniting ESC/Java and JML. In *Revised selected papers of CASSIS 2004*, volume 3362 of *Lecture Notes in Computer Science*, pages 108–128. Springer-Verlag, 2005.
- [14] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th POPL*, pages 238–252, 1977.
- [15] Adam Darvas, Reiner Hähnle, and David Sands. A theorem proving approach to analysis of secure information flow. In *2nd International Conference on Security in Pervasive Computing (SPC 2005)*, volume 3450 of *Lecture Notes in Computer Science*, pages 151–171, 2005.
- [16] Guillaume Dufay, Amy Felty, and Stan Matwin. Privacy-sensitive information flow with JML. In *Conference on Automated Deduction (CADE-20)*, volume 3632 of *LNAI*, pages 116–130. Springer-Verlag, 2005.
- [17] Matthew B. Dwyer, John Hatcliff, Matthew Hoosier, and Robby. Building your own software model checker using the Bogor extensible model checking framework. In *17th Conference on Computer-Aided Verification (CAV 2005)*, volume 3576 of *Lecture Notes in Computer Science*, pages 148–152. Springer-Verlag, 2005.
- [18] Joseph Goguen and Jose Meseguer. Security policies and security models. In *Proc. IEEE Symp. on Security and Privacy*, pages 11–20, 1982.
- [19] Christian Hammer, Jens Krinke, and Gregor Snelting. Information flow control for Java based on path conditions in dependence graphs. In *IEEE International Symposium on Secure Software Engineering (ISSSE 2006)*, pages 87–96, March 2006.
- [20] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Software verification with BLAST. In *10th SPIN Workshop on Model Checking Software (SPIN)*, volume 2648 of *Lecture Notes in Computer Science*, pages 235–239, 2003.
- [21] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *POPL*, pages 228–241. ACM Press, 1999.
- [22] David A. Naumann. From coupling relations to mated invariants for checking information flow. In *11th European Symposium on Research in Computer Security (ESORICS'06)*, volume 4189 of *Lecture Notes in Computer Science*, pages 279–296. Springer-Verlag, 2006.
- [23] Andrei Sabelfeld and Andrew C. Myers. A model for delimited information release. In *2003 International Symposium on Software Security (ISSS'03), Tokyo, Japan*, volume 3233 of *Lecture Notes in Computer Science*, pages 174–191. Springer-Verlag, 2004.
- [24] Tachio Terauchi and Alex Aiken. Secure information flow as a safety problem. In *Static Analysis Symposium (SAS)*, volume 3672 of *Lecture Notes in Computer Science*, pages 352–367, 2005.
- [25] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

A Proof of Correctness

To establish Theorem 4.2, we shall need to establish a sequence of auxiliary results, including Lemmas 4.3, 4.6, and 4.7, and Proposition 4.1. Remember that for a call $[VC]\{\Theta\} (R) \Leftarrow S\{\Theta'\}$ where S is a top-level command, we always assume that Θ' is field-free (and then prove, in Lemma 4.3, that also Θ is field-free).

A.1 Basic Results about Substitution

Lemma A.1 *For all A_0, A, s, r, x , with $v = \llbracket A \rrbracket_r^s$ and with $s' = [s \mid x \mapsto v]$, we have*

$$\llbracket A_0[A/x] \rrbracket_r^s = \llbracket A_0 \rrbracket_{r'}^{s'}.$$

Proof: Structural induction in A_0 , with a case analysis on the form of A_0 . If A_0 is a constant c , then both sides evaluate to c ; similarly if $A_0 = \text{nil}$. If $A_0 = x$, then both sides evaluate to v . If A_0 is a variable y with $y \neq x$, then both sides evaluate to $s(y)$. If A_0 is of the form $.f$, then both sides evaluate to $r(f)$. If A_0 is of the form $A_1 \text{ op } A_2$, the claim follows easily from the induction hypothesis. \square

Similarly, we can prove

Lemma A.2 *For all A_0, A, s, r, f , with $v = \llbracket A \rrbracket_r^s$ and with $r' = [r \mid f \mapsto v]$, we have*

$$\llbracket A_0[A/.f] \rrbracket_r^s = \llbracket A_0 \rrbracket_{r'}^s.$$

Lemma A.3 *For all B, A, s, r, x , with $v = \llbracket A \rrbracket_r^s$ and with $s' = [s \mid x \mapsto v]$, we have*

$$\llbracket B[A/x] \rrbracket_r^s = \llbracket B \rrbracket_{r'}^{s'}.$$

Proof: Wlog., we can assume that B is of the form $A_1 < A_2$. Then $\llbracket B[A/x] \rrbracket_r^s$ is true iff $\llbracket A_1[A/x] \rrbracket_r^s < \llbracket A_2[A/x] \rrbracket_r^s$ iff (by Lemma A.1) $\llbracket A_1 \rrbracket_{r'}^{s'} < \llbracket A_2 \rrbracket_{r'}^{s'}$ iff $\llbracket B \rrbracket_{r'}^{s'}$ is true. \square

Similarly, we can prove

Lemma A.4 *For all B, A, s, r, f , with $v = \llbracket A \rrbracket_r^s$ and with $r' = [r \mid f \mapsto v]$, we have*

$$\llbracket B[A/.f] \rrbracket_r^s = \llbracket B \rrbracket_{r'}^s.$$

As a consequence of Lemmas A.1 and A.3, we have:

Lemma A.5 *For all E, A, s, r, x , with $v = \llbracket A \rrbracket_r^s$ and with $s' = [s \mid x \mapsto v]$, we have*

$$\llbracket E[A/x] \rrbracket_r^s = \llbracket E \rrbracket_{r'}^{s'}.$$

As a consequence of Lemmas A.2 and A.4, we have:

Lemma A.6 *For all E, A, s, r, f , with $v = \llbracket A \rrbracket_r^s$ and with $r' = [r \mid f \mapsto v]$, we have*

$$\llbracket E[A/.f] \rrbracket_r^s = \llbracket E \rrbracket_{r'}^s.$$

Lemma A.7 *For all ϕ, A, s, r, x , with $v = \llbracket A \rrbracket_r^s$ and with $s' = [s \mid x \mapsto v]$, we have*

$$s, r \models \phi[A/x] \text{ iff } s', r \models \phi.$$

Proof: Structural induction in ϕ , with a case analysis on the form of ϕ . If ϕ is some boolean expression B , the claim follows directly from Lemma A.3. Otherwise, for instance when ϕ is of the form $\phi_1 \wedge \phi_2$, the claim follows easily from the induction hypothesis. \square

Similarly, we can prove

Lemma A.8 *For all ϕ, A, s, r, f , with $v = \llbracket A \rrbracket_r^s$ and with $r' = [r \mid f \mapsto v]$, we have*

$$s, r \models \phi[A/.f] \text{ iff } s, r' \models \phi.$$

Results about rm_x^ℓ . Recall that for x with $type(x) = \text{obj}$, we defined $rm_x^\ell : \mathbf{1Assert} \rightarrow \mathbf{1Assert}$ by

$$\begin{aligned}
rm_x^\ell(x = x) &= true \\
rm_x^\ell(x = A) &= false && \text{if } x \notin \text{fv}(A) \\
rm_x^\ell(A = x) &= false && \text{if } x \notin \text{fv}(A) \\
rm_x^\ell(B) &= B && \text{if } x \notin \text{fv}(B) \\
rm_x^\ell(\phi_1 \wedge \phi_2) &= rm_x^\ell(\phi_1) \wedge rm_x^\ell(\phi_2) \\
rm_x^\ell(\phi_1 \vee \phi_2) &= rm_x^\ell(\phi_1) \vee rm_x^\ell(\phi_2) \\
rm_x^\ell(\neg\phi_0) &= \neg rm_x^\ell(\phi_0)
\end{aligned}$$

where the clauses are exhaustive, due to Fact 3.1.

Lemma A.9 For all s, x, l , with $type(x) = \text{obj}$ and $l \notin \text{ran}(s)$, with $s' = [s \mid x \mapsto l]$ we have

$$s' \models \phi \text{ iff } s \models rm_x^\ell(\phi)$$

Proof: Induction in ϕ , with a case analysis on the form of ϕ ; the inductive cases are straightforward so we only look at the base cases. If ϕ is of the form $(x = x)$, both sides evaluate to True. If ϕ is of the form $(x = A)$, or $(A = x)$, with $x \notin \text{fv}(A)$, then $rm_x^\ell(\phi) = false$ so the claim boils down to $s' \models x \neq A$, that is $l \neq \llbracket A \rrbracket^s$ which follows from $l \notin \text{ran}(s)$ (and the absence of pointer arithmetic). If ϕ is of the form B with $x \notin \text{fv}(B)$ then the claim is that $s' \models B$ iff $s \models B$, which is obvious. \square

Results about rm_x^r . Recall that for x with $type(x) = \text{obj}$, we defined $rm_x^r : \mathbf{Exp} \rightarrow \mathbf{Exp}$ by

$$\begin{aligned}
rm_x^r(E) &= 0 && \text{if } x \in \text{fv}(E) \\
rm_x^r(E) &= E && \text{if } x \notin \text{fv}(E)
\end{aligned}$$

Lemma A.10 For all s, s_1, r, r_1 , and x (with $type(x) = \text{obj}$), for all E with $x \in \text{fv}(E)$, for all $l \notin \text{ran}(s)$ and $l_1 \notin \text{ran}(s_1)$, and β with $l \beta l_1$, with $s' = [s \mid x \mapsto l]$ and $s'_1 = [s_1 \mid x \mapsto l_1]$ we have

$$\llbracket E \rrbracket_r^{s'} \beta \llbracket E \rrbracket_{r_1}^{s'_1}.$$

Proof: From Fact 3.1 we infer that (apart from symmetry) there are only 3 cases. If E is x then the claim is $s'(x) \beta s'_1(x)$ which follows from our assumptions. If E is $(x = x)$ then the claim is True β True which follows from our conventions. Finally, consider the case where E is of the form $(x = A)$ with $x \notin \text{fv}(A)$. From $l \notin \text{ran}(s)$ (and the absence of pointer arithmetic) we infer $\llbracket A \rrbracket_r^s \neq l$ and hence also $\llbracket A \rrbracket_r^{s'} \neq l$; similarly we infer $\llbracket A \rrbracket_{r_1}^{s'_1} \neq l_1$. But since $\llbracket x \rrbracket_r^{s'} = l$ and $\llbracket x \rrbracket_{r_1}^{s'_1} = l_1$, the claim boils down to False β False which follows from our conventions. \square

Results about ff^+ . Recall that for $ff^+ : \mathbf{1Assert} \rightarrow \mathbf{1Assert}$, we require the following result to hold:

Fact A.11 If $\phi' = ff^+(\phi)$ then ϕ' is field-free, and ϕ logically implies ϕ' .

One way to accomplish that is given below, where ff^+ is defined simultaneously with its dual ff^- which has the property that if $\phi' = ff^-(\phi)$ then ϕ' is field-free, and ϕ' logically implies ϕ .

$$\begin{aligned}
ff^+(B) &= true && \text{if } B \text{ contains field names} \\
ff^+(B) &= B && \text{if } B \text{ is field-free} \\
ff^+(\phi_1 \wedge \phi_2) &= ff^+(\phi_1) \wedge ff^+(\phi_2) \\
ff^+(\phi_1 \vee \phi_2) &= ff^+(\phi_1) \vee ff^+(\phi_2) \\
ff^+(\neg\phi_0) &= \neg ff^-(\phi_0) \\
ff^-(B) &= false && \text{if } B \text{ contains field names} \\
ff^-(B) &= B && \text{if } B \text{ is field-free} \\
ff^-(\phi_1 \wedge \phi_2) &= ff^-(\phi_1) \wedge ff^-(\phi_2) \\
ff^-(\phi_1 \vee \phi_2) &= ff^-(\phi_1) \vee ff^-(\phi_2) \\
ff^-(\neg\phi_0) &= \neg ff^+(\phi_0)
\end{aligned}$$

A.2 Totality and Write Confinement

Lemma 4.3 Assume $[VC]\{\Theta\}(R) \Leftarrow S\{\Theta'\}$. Then

Totality $dom(R) = \Theta$ (left totality) and $ran(R) = \Theta'$ (right totality).

Wellformedness If S is a top-level command and Θ' is field-free then also Θ is field-free.

Uniqueness Given $\theta' \in \Theta'$, there exists at most one θ such that $(\theta, u, \theta') \in R$.

Write Confinement If $(\theta, u, \theta') \in R$, then $con(\theta) = con(\theta')$, and with $E = con(\theta)$ we have

- if $s, - \llbracket S \rrbracket s', -$ then s agrees with s' on $fv(E)$;
- if $s, r \llbracket S \rrbracket s', r'$ (thus S is of form RS) then also r agrees with r' on $ff(E)$.

Proof: The proof is by induction in S , with a case analysis on the form of S where we shall use the terminology from Figs. 5 and 6.

$S = \text{skip}$ or $S = \text{assert}(\phi_0)$. The claims are all trivial.

$S = x := A$. Left Totality follows from the construction of Θ , while Right Totality follows from the construction of R . Wellformedness follows since if S is a top-level command then A is field-free; Uniqueness is trivial since for each $\theta' \in \Theta'$ there exists exactly one θ with $(\theta, -, \theta') \in R$. For Write Confinement, if $(\theta, u, \theta') \in R$ with $E = con(\theta')$ then $x \notin fv(E)$, so $E[A/x] = E$ and the claim is obvious.

$S = .f := A$. This is similar to the previous case.

$S = S_1 ; S_2$. The induction hypothesis obviously gives us Totality, Wellformedness, and Uniqueness. For Write Confinement, assume that $(\theta, u, \theta') \in R$; this happens because there exists θ'' with $(\theta, u, \theta'') \in R_1$ and $(\theta'', u, \theta') \in R_2$. With $E = con(\theta')$, inductively we infer first $E = con(\theta'')$ and next $E = con(\theta)$. Finally, assume that $s, r \llbracket S \rrbracket s', r'$ (the case $s, h \llbracket S \rrbracket s', h'$ is similar); then there exists s'', r'' such that $s, r \llbracket S_1 \rrbracket s'', r''$ and $s'', r'' \llbracket S_2 \rrbracket s', r'$. Given $x \in fv(E)$, we must show that $s(x) = s'(x)$, but this follows since inductively we have $s(x) = s''(x)$ and $s''(x) = s'(x)$. Similarly, we can show that for $f \in ff(E)$ we have $r(f) = r'(f)$.

$S = \text{if } B \text{ then } S_1 \text{ else } S_2$. Wellformedness follows from the induction hypothesis, since if S is a top-level command then B is field-free. Also Uniqueness follows easily from the induction hypothesis. Left Totality follows from the construction of Θ . For Right Totality, consider $\theta' \in \Theta'$. If $\theta' \in \Theta'_m$ then the claim follows, due to R'_1 , since inductively we can assume that R_1 is total. If $\theta' \in \Theta'_u$ then we infer inductively that there exists θ_1, θ_2 such that $(\theta_1, u, \theta') \in R_1$ and $(\theta_2, u, \theta') \in R_2$; we also infer inductively that $\text{con}(\theta_1) = \text{con}(\theta') = \text{con}(\theta_2)$. But this shows, due to R_0 , that there exists θ with $(\theta, u, \theta') \in R$.

We now address Write Confinement, and consider $(\theta, u, \theta') \in R$, that is $(\theta, u, \theta') \in R_0$ with $\theta' \in \Theta'_u$. Thus there exists $(\theta_1, u, \theta') \in R_1$ and $(\theta_2, u, \theta') \in R_2$ with $\text{con}(\theta) = \text{con}(\theta_1) = \text{con}(\theta_2)$; inductively, we infer that $\text{con}(\theta_1) = \text{con}(\theta')$ and hence $\text{con}(\theta) = \text{con}(\theta')$. Finally, assume that $s, r \llbracket S \rrbracket s', r'$ (the case $s, h \llbracket S \rrbracket s', h'$ is similar) where we can assume, wlog., that $s, r \models B$ and $s, r \llbracket S_1 \rrbracket s', r'$. Given $x \in \text{fv}(E)$ (or $f \in \text{ff}(E)$), we must show that $s(x) = s'(x)$ (or $r(f) = r'(f)$), but this follows from the induction hypothesis.

$S = \text{new } x \text{ in } RS \text{ close}$. Left Totality follows from the construction of Θ , while Right Totality follows from the induction hypothesis (and the construction of R). Wellformedness follows from the construction of R , since all field names are replaced by their default values.

Now assume that $(\phi \Rightarrow E \times, u, \phi' \Rightarrow E' \times) \in R$. Then there exists $(\phi_0 \Rightarrow E_0 \times, u, \phi' \Rightarrow E' \times) \in R_0$ such that $\phi = \text{rm}_x^\ell(\phi_0[\overline{\text{deflt}(f)}/\bar{f}])$ and $E = \text{rm}_x^r(E_0[\overline{\text{deflt}(f)}/\bar{f}])$. Uniqueness now follows from the induction hypothesis. From the construction of R we also have $x \notin \text{fv}(E_0)$. To establish Write Confinement, first observe that inductively we have $E_0 = E'$ which is field-free by our overall assumption; thus $E = \text{rm}_x^r(E_0)$ which since $x \notin \text{fv}(E_0)$ implies $E = E_0 = E'$. Finally, assume that $s, h \llbracket S \rrbracket s', h'$ so as to show that s and s' agree on $\text{fv}(E)$. We have $[s \mid x \mapsto l]_{,-} \llbracket RS \rrbracket s',_{-}$ so inductively on RS we can assume that $[s \mid x \mapsto l]$ and s' agree on $\text{fv}(E_0)$; this is as desired since $x \notin \text{fv}(E_0)$.

$S = \text{open } x \text{ in } RS \text{ close}$. Left Totality follows from the construction of Θ . Right Totality follows from the induction hypothesis, due to R_1 and R_4 . Concerning Wellformedness, the only issue (since ff^+ produces field-free assertions) is whether E as mentioned in R_4 is field-free. But inductively on RS , Write Confinement tells us that in this case, $E = \text{con}(\theta')$ where θ' is field-free by assumption.

Now assume that $(\phi \Rightarrow E \times, u, \theta') \in R$. We infer that $(\phi \Rightarrow E \times, u, \theta') \in R_4$, and that there exists $(\phi_0 \Rightarrow E \times, u, \theta') \in R_0$ such that $\phi = \text{ff}^+(\phi_0)$. Uniqueness now follows from the induction hypothesis. To establish Write Confinement, first observe that $E = \text{con}(\theta')$ follows from the induction hypothesis. Finally, assume that $s, h \llbracket S \rrbracket s', h'$ so as to show that s and s' agree on $\text{fv}(E)$; since there exists r, r' such that $s, r \llbracket RS \rrbracket s', r'$, the claim follows from the induction hypothesis applied to RS .

$S = \text{while } B \text{ do } S_0$. (We only consider the case where S is a top-level command, the other case being similar.) Totality, Wellformedness, and Uniqueness are trivial. Now assume that $(\theta, u, \theta') \in R$. We infer $\theta = \theta'$; let $E = \text{con}(\theta)$. Since $\theta \in \Theta_u$ there exists no $(-, m, \theta) \in R_0$. Inductively on S_0 , we thus infer by Totality that there exists $(-, u, \theta) \in R_0$, and next by Write Confinement that if $s, h \llbracket S_0 \rrbracket s', h'$ then s and s' agree on $\text{fv}(E)$. With f_i as defined in Fig. 4, it is now easy to show by induction in i that if $s, h f_i s', h'$ then s and s' agree on $\text{fv}(E)$. But this establishes Write Confinement. \square

A.3 Other Key Lemmas

Lemma 4.6 Assume $[VC]\{\Theta\} (R) \Leftarrow S \{\Theta'\}$ where $\models VC$ holds. Given $\theta' \in \Theta'$, there exists $(\theta, -, \theta') \in R$ such that

- if $S = RS$: whenever $s, r \llbracket S \rrbracket s', r'$ and $s', r' \models \text{ant}(\theta')$ then $s, r \models \text{ant}(\theta)$;

- if $S = TS$: whenever $s, h \llbracket S \rrbracket s', h'$ and $s' \models \text{ant}(\theta')$ then $s \models \text{ant}(\theta)$.

For $S = \mathbf{while} B \mathbf{do} S_0$, if $\theta' \in \Theta_u$ we can use $\theta = \theta'$, otherwise we can use $\theta = \theta_m$.

Proof: We define $Q(S, \theta, \theta')$ as the following property:

whenever $s, h \llbracket S \rrbracket s', h'$ (if S is a top-level command) or $s, r \llbracket S \rrbracket s', r'$ (otherwise),
and $s' \models \text{ant}(\theta')$ (if S is a top-level command) or $s', r' \models \text{ant}(\theta')$ (otherwise),
then $s \models \text{ant}(\theta)$ (if S is a top-level command) or $s, r \models \text{ant}(\theta)$ (otherwise).

The claim is thus that if $[VC]\{\Theta\} (R) \Leftarrow S \{\Theta'\}$ and $\models VC$ holds, then for all $\theta' \in \Theta'$ there exists $(\theta, _, \theta') \in R$ with $Q(S, \theta, \theta')$ (and that for a while loop, θ is given explicitly in a certain way).

The proof is by induction in S , with a case analysis on the form of S where we shall use the terminology from Figs. 5 and 6.

$S = \mathbf{skip}$. Trivial.

$S = \mathbf{assert}(\phi_0)$. Given $\theta' \in \Theta'$, there exists $(\theta, _, \theta') \in R$ such that $\text{ant}(\theta) = \text{ant}(\theta') \wedge \phi_0$. We shall now show $Q(S, \theta, \theta')$, for the case when S is a record command (the other case is similar). So assume $s, r \llbracket RS \rrbracket s', r'$, which amounts to $s, r \models \phi_0$ and $s' = s$ and $r' = r$, and $s', r' \models \text{ant}(\theta')$. But then clearly $s, r \models \text{ant}(\theta)$.

$S = x := A$. Given $\theta' \in \Theta'$, there exists $(\theta, _, \theta') \in R$ such that with $\phi = \text{ant}(\theta)$ and $\phi' = \text{ant}(\theta')$ we have $\phi = \phi'[A/x]$. We shall now show $Q(S, \theta, \theta')$, for the case when S is a record command (the other case is similar). So assume $s, r \llbracket S \rrbracket s', r'$, which amounts to $r' = r$ and $s' = [s \mid x \mapsto v]$ where $v = \llbracket A \rrbracket_r^s$, and that $s', r' \models \phi'$. But by Lemma A.7, this implies the desired $s, r \models \phi$.

$S = .f := A$. Given $\theta' \in \Theta'$, there exists $(\theta, _, \theta') \in R$ such that with $\phi = \text{ant}(\theta)$ and $\phi' = \text{ant}(\theta')$ we have $\phi = \phi'[A/.f]$. Here S is a record command, and we shall now show $Q(S, \theta, \theta')$. So assume $s, r \llbracket S \rrbracket s', r'$, which amounts to $s' = s$ and $r' = [r \mid f \mapsto v]$ where $v = \llbracket A \rrbracket_r^s$, and that $s', r' \models \phi'$. But by Lemma A.8, this implies the desired $s, r \models \phi$.

$S = S_1 ; S_2$. Given $\theta' \in \Theta'$, inductively on S_2 we can find $(\theta'', _, \theta') \in R_2$ such that $Q(S_2, \theta'', \theta')$, and inductively on S_1 we can next find $(\theta, _, \theta'') \in R_1$ such that $Q(S_1, \theta, \theta'')$. Since $(\theta, _, \theta') \in R$, our task can be accomplished by showing $Q(S, \theta, \theta')$, where we only consider the case where S is a record command (as the other case is similar). So assume $s, r \llbracket S \rrbracket s', r'$, that is there exists s'', r'' with $s, r \llbracket S_1 \rrbracket s'', r''$ and $s'', r'' \llbracket S_2 \rrbracket s', r'$, and that $s', r' \models \text{ant}(\theta')$. From $Q(S_2, \theta'', \theta')$ we now infer $s'', r'' \models \text{ant}(\theta'')$, and from $Q(S_1, \theta, \theta'')$ we next infer the desired $s, r \models \text{ant}(\theta)$.

$S = \mathbf{if} B \mathbf{then} S_1 \mathbf{else} S_2$. Given $\theta' \in \Theta'$, inductively on S_1 we can find $(\theta_1, _, \theta') \in R_1$ such that $Q(S_1, \theta_1, \theta')$, and inductively on S_2 we can find $(\theta_2, _, \theta') \in R_2$ such that $Q(S_2, \theta_2, \theta')$. Let $\theta_1 = (\phi_1 \Rightarrow E_1 \times)$, let $\theta_2 = (\phi_2 \Rightarrow E_2 \times)$, and let $\theta' = (\phi' \Rightarrow E \times)$. We now first define ϕ as $(\phi_1 \wedge B) \vee (\phi_2 \wedge \neg B)$, and next define θ as follows: if $\theta' \in \Theta'_m$ then we let $\theta = (\phi \Rightarrow B \times)$ so that $(\theta, _, \theta') \in R'_0 \subseteq R$; if $\theta' \in \Theta'_u$, in which case Lemma 4.3 tells us that $E = E_1 = E_2$, we let $\theta = (\phi \Rightarrow E \times)$ so that $(\theta, _, \theta') \in R_0 \subseteq R$. We are left with proving $Q(S, \theta, \theta')$ where we only consider the case where S is a record command (as the other case is similar). So assume $s, r \llbracket S \rrbracket s', r'$, where wlog. we may assume that $s, r \models B$ and $s, r \llbracket S_1 \rrbracket s', r'$, and that $s', r' \models \phi'$. From $Q(S_1, \theta_1, \theta')$ we now infer $s, r \models \phi_1$, and thus $s, r \models \phi_1 \wedge B$ implying the desired $s, r \models \phi$.

$S = \mathbf{new} x \mathbf{in} RS \mathbf{close}$. Given $\theta' \in \Theta'$, inductively on RS we can find $(\theta_0, _, \theta') \in R_0$ such that $Q(RS, \theta_0, \theta')$. Let $\phi' = \text{ant}(\theta')$ and $\phi_0 = \text{ant}(\theta_0)$, and let $\phi = \text{rm}_x^{\ell}(\phi_0[\text{deflt}(f)/f])$. The algorithm

now yields θ with $\phi = \text{ant}(\theta)$ such that $(\theta, -, \theta') \in R$; we shall prove $Q(S, \theta, \theta')$. So assume $s, h \llbracket S \rrbracket s', h'$, implying that with $r_0 = \text{deflt}$ there exists l with $l \notin \text{ran}(s)$ such that $[s \mid x \mapsto l], r_0 \llbracket RS \rrbracket s', -,$ and assume $s' \models \phi'$. From $Q(RS, \theta_0, \theta')$ we now infer $[s \mid x \mapsto l], r_0 \models \phi_0$, and by repeated applications of Lemma A.8 we next infer that $[s \mid x \mapsto l] \models \phi_0[\text{deflt}(f)/f]$. By Lemma A.9, this implies the desired $s \models \phi$.

$S = \text{open } x \text{ in } RS \text{ close}$. Given $\theta' \in \Theta'$, inductively on RS we can find $(\theta_0, -, \theta') \in R_0$ such that $Q(RS, \theta_0, \theta')$. Let $\phi' = \text{ant}(\theta')$ and $\phi_0 = \text{ant}(\theta_0)$. The algorithm, either through R_1 or R_4 , now yields θ with $\text{ant}(\theta) = \text{ff}^+(\phi_0)$ such that $(\theta, -, \theta') \in R$; we shall prove $Q(S, \theta, \theta')$. So assume $s, h \llbracket S \rrbracket s', h'$, implying that $s, h(l) \llbracket RS \rrbracket s', h'(l)$, and assume $s' \models \phi'$. From $Q(RS, \theta_0, \theta')$ we now infer $s, h(l) \models \phi_0$ which by Fact A.11 implies the desired $s \models \text{ant}(\theta)$.

$S = \text{while } B \text{ do } S_0$. We only consider the case when S is a top-level command, as the other case is similar. Given $\theta' \in \Theta'$, first consider the case when $\theta' \in \Theta_u$. Then $(\theta', -, \theta') \in R$, so with $\phi' = \text{ant}(\theta')$ and with f_i as defined in Fig. 4 it is sufficient to prove for all i that if $s, h f_i s', h'$ then $s' \models \phi'$ implies $s \models \phi'$. We shall do so by an inner induction in i . For $i = 0$, we have $s = s'$ and the claim is trivial. Otherwise, we have $s, h \llbracket S_0 \rrbracket s'', h''$ and $s'', h'' f_{i-1} s', h'$. Now assume $s' \models \phi'$. By the inner induction hypothesis, we have $s'' \models \phi'$. Applying the outermost induction hypothesis on S_0 , we find $(\theta_0, -, \theta') \in R_0$ such that $s \models \phi_0$ where $\phi_0 = \text{ant}(\theta_0)$. But $(\phi_0 \triangleright^1 \phi') \in VC_5 \subseteq VC$, so from $\models VC$ we infer the desired $s \models \phi'$.

Next consider the case when $\theta' \in \Theta_m$. Then $(\theta_m, -, \theta') \in R$. Let $\phi' = \text{ant}(\theta')$ and $\phi_m = \text{ant}(\theta_m)$. Since $(\phi' \triangleright^1 \phi_m) \in VC_3 \subseteq VC$, we from $\models VC$ infer that $s' \models \phi'$ implies $s' \models \phi_m$. It is thus sufficient to prove that if $s, h f_i s', h'$ then $s' \models \phi_m$ implies $s \models \phi_m$. We shall do so by an inner induction in i . For $i = 0$, we have $s = s'$ and the claim is trivial. Otherwise, we have $s, h \llbracket S_0 \rrbracket s'', h''$ and $s'', h'' f_{i-1} s', h'$. Now assume $s' \models \phi_m$. By the inner induction hypothesis, we have $s'' \models \phi_m$. Applying the outermost induction hypothesis on S_0 , we find $(\theta_0, -, \theta_m) \in R_0$ such that $s \models \phi_0$ where $\phi_0 = \text{ant}(\theta_0)$. But $(\phi_0 \triangleright^1 \phi_m) \in VC_4 \subseteq VC$, so from $\models VC$ we infer the desired $s \models \phi_m$. \square

Lemma 4.7 Assume $[VC]\{\Theta\} (R) \Leftarrow TS \{\Theta'\}$ where $\models VC$ holds, and that there exists $\theta' \in \Theta'$ such that if $(-, \gamma, \theta') \in R$ then $\gamma = u$. For $(\phi_0 \Rightarrow E_0 \times) \in \mathcal{I}$, if $s, h \llbracket TS \rrbracket s', h'$ then for all $l \in \text{dom}(h)$

- if $h'(l) \models \phi_0$ then $h(l) \models \phi_0$;
- $h(l)(f) = h'(l)(f)$ for all f in $\text{ff}(E_0)$.

Proof: We assume that $(\phi_0 \Rightarrow E_0 \times) \in \mathcal{I}$ has been given, and define $Q(T)$ as the following property (here T relates states to states):

- if $s, h T s', h'$ then for all $l \in \text{dom}(h)$, for all $f \in \text{ff}(E_0)$:
- (i) if $h'(l) \models \phi_0$ then $h(l) \models \phi_0$;
 - (ii) $h(l)(f) = h'(l)(f)$.

The claim of the lemma is thus that if $[VC]\{\Theta\} (R) \Leftarrow TS \{\Theta'\}$ where $\models VC$ and where for some $\theta' \in \Theta'$ there is no $(-, m, \theta') \in R$, then $Q(\llbracket TS \rrbracket)$.

We shall prove this claim by induction in TS , with a case analysis on the form of TS where we shall use the terminology from Figs. 5 and 6. Some cases are trivial since $h' = h$ holds there: $TS = \text{skip}$, $TS = \text{assert}(\phi)$, $TS = x := A$.

$TS = TS_1 ; TS_2$. Our assumption is that there exists $\theta' \in \Theta'$ such that R contains no $(-, m, \theta')$. By Lemma 4.3 (Totality) applied to TS_1 , this implies that R_2 contains no $(-, m, \theta')$, but by Lemma 4.3 (Totality) applied to TS_2 , we see that there exists $\theta'' \in \Theta''$ such that $(\theta'', u, \theta') \in R_2$, from which we infer

that R_1 contains no $(-, m, \theta'')$. Since we also have $\models VC_1$ and $\models VC_2$ (from our assumption $\models VC$), the induction hypothesis tells us that $Q(\llbracket TS_1 \rrbracket)$ and $Q(\llbracket TS_2 \rrbracket)$.

We shall now prove $Q(\llbracket TS \rrbracket)$, so assume $s, h \llbracket TS \rrbracket s', h'$, implying that for some s'', h'' we have $s, h \llbracket TS_1 \rrbracket s'', h''$ and $s'', h'' \llbracket TS_2 \rrbracket s', h'$. Let $l \in \text{dom}(h)$ ($\subseteq \text{dom}(h'')$) be given. If $h'(l) \models \phi_0$ then from $Q(\llbracket TS_2 \rrbracket)$ we get $h''(l) \models \phi_0$ and from $Q(\llbracket TS_1 \rrbracket)$ then $h(l) \models \phi_0$. Finally, given $f \in \text{ff}(E_0)$, from $Q(\llbracket TS_1 \rrbracket)$ we get $h(l)(f) = h''(l)(f)$ and from $Q(\llbracket TS_2 \rrbracket)$ we get $h''(l)(f) = h'(l)(f)$, yielding the desired $h(l)(f) = h'(l)(f)$.

$TS = \text{if } B \text{ then } TS_1 \text{ else } TS_2$. Our assumption is that there exists $\theta' \in \Theta'$ such that R contains no $(-, m, \theta')$. We infer that $\theta' \in \Theta'_u$, and therefore neither R_1 nor R_2 contains any $(-, m, \theta')$. Since we also have $\models VC_1$ and $\models VC_2$ (from our assumption $\models VC$), the induction hypothesis tells us that $Q(\llbracket TS_1 \rrbracket)$ and $Q(\llbracket TS_2 \rrbracket)$.

We shall now prove $Q(\llbracket TS \rrbracket)$, so assume $s, h \llbracket TS \rrbracket s', h'$. Wlog., we can assume that $s, h \llbracket TS_1 \rrbracket s', h'$, and the claim now follows from $Q(\llbracket TS_1 \rrbracket)$.

$TS = \text{while } B \text{ do } TS_0$. Our assumption is that there exists $\theta' \in \Theta'$ such that R contains no $(-, m, \theta')$. We infer that $\theta' \in \Theta'_u$, and therefore R_0 contains no $(-, m, \theta')$. Since we also have $\models VC_0$ (from our assumption $\models VC$), the induction hypothesis tells us that $Q(\llbracket TS_0 \rrbracket)$.

It is sufficient to prove that $Q(f_i)$ holds for all i , which we shall do by induction in i . So assume $s, h f_i s', h'$. For $i = 0$, the claim is obvious as then $h' = h$. Otherwise, there exists s'', h'' such that $s, h \llbracket TS_0 \rrbracket s'', h''$ and $s'', h'' \llbracket f_{i-1} \rrbracket s', h'$. Let $l \in \text{dom}(h)$ ($\subseteq \text{dom}(h'')$) be given. If $h'(l) \models \phi_0$ then from the inner induction hypothesis we get $h''(l) \models \phi_0$ and from $Q(\llbracket TS_0 \rrbracket)$ then $h(l) \models \phi_0$. Finally, given $f \in \text{ff}(E_0)$, from $Q(\llbracket TS_0 \rrbracket)$ we get $h(l)(f) = h''(l)(f)$ and from the inner induction hypothesis we get $h''(l)(f) = h'(l)(f)$, yielding the desired $h(l)(f) = h'(l)(f)$.

$TS = \text{new } x \text{ in } RS \text{ close}$. Here the claim is trivial since if $l \in \text{dom}(h)$ then $h'(l) = h(l)$.

$TS = \text{open } x \text{ in } RS \text{ close}$. Our assumption is that $\models VC$ and that for some $\theta' \in \Theta'$, there exists no $(-, m, \theta') \in R$. Due to R_2 , we therefore infer that there exists no $(-, m, \phi_0 \Rightarrow E_0 \times) \in R_0$. We can now apply Lemma 4.3 to RS and infer that there exists exactly one θ_0 such that $(\theta_0, u, \phi_0 \Rightarrow E_0 \times) \in R_0$.

To prove $Q(\llbracket TS \rrbracket)$, we assume $s, h \llbracket S \rrbracket s', h'$. Thus there exists r' such that with $l_0 = s(x)$ and $r = h(l_0)$ we have $s, r \llbracket RS \rrbracket s', r'$ where $h' = [h \mid l_0 \mapsto r']$. Let $l \in \text{dom}(h)$ be given. If $l \neq l_0$ the claims are obvious, since then $h'(l) = h(l)$. So assume that $l = l_0$, leaving us with two obligations.

First, we must prove that for all $f \in \text{ff}(E_0)$, $r(f) = r'(f)$. But this follows from applying Lemma 4.3 to RS , using $(\theta_0, u, \phi_0 \Rightarrow E_0 \times) \in R_0$.

Last, we must prove that if $r' \models \phi_0$ then $r \models \phi_0$. But assuming $r' \models \phi_0$, we get $s, r \models \text{ant}(\theta_0)$ by applying Lemma 4.6 to RS , using that θ_0 is unique with $(\theta_0, u, \phi_0 \Rightarrow E_0 \times) \in R_0$. Since $(\text{ant}(\theta_0) \triangleright^1 \phi_0) \in VC_1 \subseteq VC$, from $\models VC$ we infer $s, r \models \phi_0$ which (since ϕ_0 is an object assertion) amounts to the desired $r \models \phi_0$. \square

A.4 Correctness of Record Commands

Proposition 4.1 Assume that

1. $[VC]\{\Theta\} (-) \Leftarrow RS \{\Theta'\}$ and that $\models VC$
2. $s, r \llbracket RS \rrbracket s', r'$ and $s_1, r_1 \llbracket RS \rrbracket s'_1, r'_1$
3. $s, r \& s_1, r_1 \models_\beta \Theta$.

Then $s', r' \& s'_1, r'_1 \models_{\beta} \Theta'$.

Proof: The proof is by induction in RS , with a case analysis on the form of RS where we shall use the terminology from Figs. 5 and 6.

$RS = \text{skip}$. Trivial.

$RS = \text{assert}(\phi_0)$. We have $s, r \models \phi_0$, $s_1, r_1 \models \phi_0$, $r' = r$, $s' = s$, $r'_1 = r_1$, $s'_1 = s_1$. Let $(\phi \Rightarrow E \times) \in \Theta'$ be given, and assume that $s, r \models \phi$ and $s_1, r_1 \models \phi$; we must prove $\llbracket E \rrbracket_r^s \beta \llbracket E \rrbracket_{r'_1}^{s'_1}$. But $(\phi \wedge \phi_0) \Rightarrow E \times \in \Theta$ so from $s, r \& s_1, r_1 \models_{\beta} \Theta$ we have $s, r \& s_1, r_1 \models_{\beta} (\phi \wedge \phi_0) \Rightarrow E \times$, and since $s, r \models \phi \wedge \phi_0$ and $s_1, r_1 \models \phi \wedge \phi_0$ this implies the desired $\llbracket E \rrbracket_r^s \beta \llbracket E \rrbracket_{r'_1}^{s'_1}$.

$RS = RS_1 ; RS_2$. There exists s'', r'' and s''_1, r''_1 such that $s, r \llbracket RS_1 \rrbracket s'', r''$ and $s'', r'' \llbracket RS_2 \rrbracket s', r'$ and $s_1, r_1 \llbracket RS_1 \rrbracket s''_1, r''_1$ and $s''_1, r''_1 \llbracket RS_2 \rrbracket s'_1, r'_1$. Given $s, r \& s_1, r_1 \models_{\beta} \Theta$, we can apply the induction hypothesis to RS_1 to give $s'', r'' \& s''_1, r''_1 \models_{\beta} \Theta'$, and next apply the induction hypothesis to RS_2 to give the desired $s', r' \& s'_1, r'_1 \models_{\beta} \Theta'$.

$RS = x := A$. Given $\theta' = (\phi' \Rightarrow E' \times) \in \Theta'$; assuming $s', r' \models \phi'$ and $s'_1, r'_1 \models \phi'$ we must prove $\llbracket E' \rrbracket_{r'}^{s'} \beta \llbracket E' \rrbracket_{r'_1}^{s'_1}$. Here $s' = [s \mid x \mapsto v]$ with $v = \llbracket A \rrbracket_r^s$, $r' = r$; $s'_1 = [s_1 \mid x \mapsto v_1]$ with $v_1 = \llbracket A \rrbracket_{r_1}^{s_1}$, $r'_1 = r_1$. With $\phi = \phi'[A/x]$ and $E = E'[A/x]$ we have $\phi \Rightarrow E \times \in \Theta$ and thus $s, r \& s_1, r_1 \models_{\beta} \phi \Rightarrow E \times$. Since $s, r \models \phi$ and $s_1, r_1 \models \phi$ follow by Lemma A.7 from $s', r' \models \phi'$ and $s'_1, r'_1 \models \phi'$, we infer $\llbracket E \rrbracket_r^s \beta \llbracket E \rrbracket_{r_1}^{s_1}$. But this amounts to the desired $\llbracket E' \rrbracket_{r'}^{s'} \beta \llbracket E' \rrbracket_{r'_1}^{s'_1}$, since by Lemma A.5 we have $\llbracket E' \rrbracket_{r'}^{s'} = \llbracket E \rrbracket_r^s$ and $\llbracket E' \rrbracket_{r'_1}^{s'_1} = \llbracket E \rrbracket_{r_1}^{s_1}$.

$RS = .f := A$. Given $\theta' = (\phi' \Rightarrow E' \times) \in \Theta'$; assuming $s', r' \models \phi'$ and $s'_1, r'_1 \models \phi'$ we must prove $\llbracket E' \rrbracket_{r'}^{s'} \beta \llbracket E' \rrbracket_{r'_1}^{s'_1}$. Here $r' = [r \mid f \mapsto v]$ with $v = \llbracket A \rrbracket_r^s$, $s' = s$; $r'_1 = [r_1 \mid f \mapsto v_1]$ with $v_1 = \llbracket A \rrbracket_{r_1}^{s_1}$, $s'_1 = s_1$. With $\phi = \phi'[A/.f]$ and $E = E'[A/.f]$ we have $\phi \Rightarrow E \times \in \Theta$ and thus $s, r \& s_1, r_1 \models_{\beta} \phi \Rightarrow E \times$. Since $s, r \models \phi$ and $s_1, r_1 \models \phi$ follow by Lemma A.8 from $s', r' \models \phi'$ and $s'_1, r'_1 \models \phi'$, we infer $\llbracket E \rrbracket_r^s \beta \llbracket E \rrbracket_{r_1}^{s_1}$. But this amounts to the desired $\llbracket E' \rrbracket_{r'}^{s'} \beta \llbracket E' \rrbracket_{r'_1}^{s'_1}$, since by Lemma A.6 we have $\llbracket E' \rrbracket_{r'}^{s'} = \llbracket E \rrbracket_r^s$ and $\llbracket E' \rrbracket_{r'_1}^{s'_1} = \llbracket E \rrbracket_{r_1}^{s_1}$.

$RS = \text{if } B \text{ then } RS_1 \text{ else } RS_2$. Except for symmetry, there are two cases. The first case is when $\llbracket B \rrbracket_r^s = \llbracket B \rrbracket_{r_1}^{s_1} = \text{True}$, in which case we have $s, r \llbracket RS_1 \rrbracket s', r'$ and $s_1, r_1 \llbracket RS_1 \rrbracket s'_1, r'_1$. It is sufficient to establish $s, r \& s_1, r_1 \models_{\beta} \Theta_1$, since then the desired $s', r' \& s'_1, r'_1 \models_{\beta} \Theta'$ will follow by induction on RS_1 . So given $(\phi_1 \Rightarrow E_1 \times) \in \Theta_1$, and assuming $s, r \models \phi_1$ and $s_1, r_1 \models \phi_1$, our obligation is to show $\llbracket E_1 \rrbracket_r^s \beta \llbracket E_1 \rrbracket_{r_1}^{s_1}$. By Lemma 4.3 (Totality) applied to RS_1 , there exists $\theta' \in \Theta'$ such that $(\phi_1 \Rightarrow E_1 \times, \neg, \theta') \in R_1$. Two cases:

- if $\theta' \in \Theta'_m$ then, by R'_1 , $(\phi_1 \wedge B \Rightarrow E_1 \times) \in \Theta$.
- if $\theta' \in \Theta'_u$ then, by R_0 and Lemma 4.3, there exists ϕ_2 such that $(\phi_1 \wedge B) \vee (\phi_2 \wedge \neg B) \Rightarrow E_1 \times \in \Theta$.

Since $s, r \models \phi_1 \wedge B$ and $s_1, r_1 \models \phi_1 \wedge B$, and since $s, r \& s_1, r_1 \models_{\beta} \Theta$, in both cases we can infer the desired $\llbracket E_1 \rrbracket_r^s \beta \llbracket E_1 \rrbracket_{r_1}^{s_1}$.

The second case to be considered is when $\llbracket B \rrbracket_r^s = \text{False}$ but $\llbracket B \rrbracket_{r_1}^{s_1} = \text{True}$, in which case we have $s, r \llbracket RS_2 \rrbracket s', r'$ and $s_1, r_1 \llbracket RS_1 \rrbracket s'_1, r'_1$. Given $\theta' = (\phi' \Rightarrow E \times) \in \Theta'$, and assuming $s', r' \models \phi'$ and $s'_1, r'_1 \models \phi'$, our proof obligation is to show $\llbracket E \rrbracket_{r'}^{s'} \beta \llbracket E \rrbracket_{r'_1}^{s'_1}$.

We shall establish that $\theta' \in \Theta'_u$, by showing that $\theta' \in \Theta'_m$ leads to a contradiction: by Lemma 4.6 applied to RS_1 and RS_2 , there exists $(\phi_1 \Rightarrow \neg \times, \neg, \theta') \in R_1$ and $(\phi_2 \Rightarrow \neg \times, \neg, \theta') \in R_2$ such that

$s_1, r_1 \models \phi_1$ and $s, r \models \phi_2$. Due to R'_0 we see that Θ contains $(\phi_1 \wedge B) \vee (\phi_2 \wedge \neg B) \Rightarrow B \times$. Since $s, r \models (\phi_1 \wedge B) \vee (\phi_2 \wedge \neg B)$ and $s_1, r_1 \models (\phi_1 \wedge B) \vee (\phi_2 \wedge \neg B)$, from $s, r \& s_1, r_1 \models_{\beta} \Theta$ we infer $\llbracket B \rrbracket_r^s \beta \llbracket B \rrbracket_{r_1}^{s_1}$. But this contradicts $\llbracket B \rrbracket_r^s \neq \llbracket B \rrbracket_{r_1}^{s_1}$.

We have established $\theta' \in \Theta'_u$. By Lemma 4.3 we now infer that there exists unique $\theta \in \Theta$ such that $(\theta, \neg, \theta') \in R$ and also that $E = \text{con}(\theta)$. By Lemma 4.6, with $\phi = \text{ant}(\theta)$, we infer that $s, r \models \phi$ and $s_1, r_1 \models \phi$. From $s, r \& s_1, r_1 \models_{\beta} \Theta$ we thus infer $\llbracket E \rrbracket_r^s \beta \llbracket E \rrbracket_{r_1}^{s_1}$. But since Lemma 4.3 also states that s, s' agree on $\text{fv}(E)$, that s_1, s'_1 agree on $\text{fv}(E)$, that r, r' agree on $\text{ff}(E)$, and that r_1, r'_1 agree on $\text{ff}(E)$, this amounts to the desired $\llbracket E \rrbracket_{r'}^{s'} \beta \llbracket E \rrbracket_{r'_1}^{s'_1}$.

$RS = \text{while } B \text{ do } RS_0$. It is sufficient to prove the following result: assume $s, r \& s_1, r_1 \models_{\beta} \Theta$, and assume that (with f_i as defined in Fig. 4) there exists i, j such that $s, r \models f_i$, $s', r' \models f_i$ and $s_1, r_1 \models f_j$, $s'_1, r'_1 \models f_j$. Then $s', r' \& s'_1, r'_1 \models_{\beta} \Theta$.

We shall prove this result by induction in $i + j$. If $i = j = 0$, the claim is obvious, as then $s' = s$, $r' = r$, $s'_1 = s_1$, $r'_1 = r_1$. Apart from symmetry, there are two other cases.

First assume $i > 0, j > 0$. Thus $\llbracket B \rrbracket_r^s = \text{True} = \llbracket B \rrbracket_{r_1}^{s_1}$, and there exists s'', r'', s''_1, r''_1 such that $s, r \llbracket RS_0 \rrbracket s'', r''$ and $s'', r'' \models f_{i-1}$, $s', r' \models f_{i-1}$ and $s_1, r_1 \llbracket RS_0 \rrbracket s''_1, r''_1$ and $s''_1, r''_1 \models f_{j-1}$, $s'_1, r'_1 \models f_{j-1}$. It is sufficient to prove

$$s, r \& s_1, r_1 \models_{\beta} \Theta_0 \quad (1)$$

since then we can first apply the outermost induction hypothesis to RS_0 , yielding $s'', r'' \& s''_1, r''_1 \models_{\beta} \Theta$, and next apply the innermost induction hypothesis, yielding the desired $s', r' \& s'_1, r'_1 \models_{\beta} \Theta$.

To prove (1), assume that $(\phi_0 \Rightarrow E_0 \times) \in \Theta_0$, and that $s, r \models \phi_0$ and $s_1, r_1 \models \phi_0$; we must show $\llbracket E_0 \rrbracket_r^s \beta \llbracket E_0 \rrbracket_{r_1}^{s_1}$. Since $(\Theta \triangleright^2 \phi_0 \wedge B \Rightarrow E_0 \times) \in VC_1 \subseteq VC$, we from $\models VC$ infer $s, r \& s_1, r_1 \models_{\beta} \phi_0 \wedge B \Rightarrow E_0 \times$, and the claim follows as $s, r \models \phi_0 \wedge B$ and $s_1, r_1 \models \phi_0 \wedge B$.

Next assume $i > 0, j = 0$. Then $\llbracket B \rrbracket_r^s = \text{True}$ and $\llbracket B \rrbracket_{r_1}^{s_1} = \text{False}$, so $s'_1 = s_1$ and $r'_1 = r_1$. To show $s', r' \& s_1, r_1 \models_{\beta} \Theta$, we consider $\theta \in \Theta$ with $\theta = (\phi \Rightarrow E \times)$, and assume that $s', r' \models \phi$ and $s_1, r_1 \models \phi$; we must show $\llbracket E \rrbracket_{r'}^{s'} \beta \llbracket E \rrbracket_{r_1}^{s_1}$. It is sufficient to prove that $\theta \in \Theta_u$ because then Lemma 4.6, applied to RS , tells us that $s, r \models \phi$ so from $s, r \& s_1, r_1 \models_{\beta} \theta$ we infer $\llbracket E \rrbracket_r^s \beta \llbracket E \rrbracket_{r_1}^{s_1}$; this is as desired since by Lemma 4.3 applied to RS , s, s' agree on $\text{fv}(E)$ and r, r' agree on $\text{ff}(E)$.

To prove $\theta \in \Theta_u$ we assume, in order to get a contradiction, that $\theta \in \Theta_m$. Lemma 4.6, applied to RS , then tells us that $s, r \models \phi_m$. Since $(\phi \triangleright^1 \phi_m) \in VC_3 \subseteq VC$, from $\models VC$ we infer that $s_1, r_1 \models \phi_m$. Since $(\Theta \triangleright^2 \phi_m \Rightarrow B \times) \in VC_2 \subseteq VC$, from $\models VC$ and $s, r \& s_1, r_1 \models_{\beta} \Theta$ we infer that $s, r \& s_1, r_1 \models_{\beta} \phi_m \Rightarrow B \times$, and thus $\llbracket B \rrbracket_r^s \beta \llbracket B \rrbracket_{r_1}^{s_1}$ which is a contradiction as $\text{True} \beta \text{False}$ cannot hold. \square

A.5 Correctness of Top-level Commands

Theorem 4.2 Assume that

1. $\llbracket VC \rrbracket \{ \Theta \} (-) \Leftarrow TS \{ \Theta' \}$ and that $\models VC$
2. $s, h \llbracket RS \rrbracket s', h'$ and that $s_1, h_1 \llbracket RS \rrbracket s'_1, h'_1$
3. $s \& s_1 \models_{\beta} \Theta$ and $h \& h_1 \models_{\beta} \mathcal{I}$.
4. There exists $\theta'_0 \in \Theta'$ such that $s' \models \text{ant}(\theta'_0)$ and $s'_1 \models \text{ant}(\theta'_0)$.

Then there exists β' extending β such that $s' \& s'_1 \models_{\beta'} \Theta'$ and $h' \& h'_1 \models_{\beta'} \mathcal{I}$.

Proof: The proof is by induction in TS , with a case analysis on the form of TS where we shall use the terminology from Figs. 5 and 6. Note that by our overall assumption, Θ' is field-free; hence also Θ is field-free, by Lemma 4.3.

$TS = \text{skip}$. Trivial, with $\beta' = \beta$.

$TS = \text{assert}(\phi_0)$. We shall prove the claim with $\beta' = \beta$, and (since $h' = h$ and $h'_1 = h_1$) the only non-trivial part is $s' \& s'_1 \models_{\beta'} \Theta'$. Here $s' = s$ and $s'_1 = s_1$; also we have $s \models \phi_0$ and $s_1 \models \phi_0$. Let $(\phi \Rightarrow E \times) \in \Theta'$ be given, and assume that $s \models \phi$ and $s_1 \models \phi$; we must prove $\llbracket E \rrbracket^s \beta \llbracket E \rrbracket^{s_1}$. But $(\phi \wedge \phi_0) \Rightarrow E \times \in \Theta$ so from $s \& s_1 \models_{\beta} \Theta$ we have $s \& s_1 \models_{\beta} (\phi \wedge \phi_0) \Rightarrow E \times$, and since $s \models \phi \wedge \phi_0$ and $s_1 \models \phi \wedge \phi_0$ this implies the desired $\llbracket E \rrbracket^s \beta \llbracket E \rrbracket^{s_1}$.

$TS = TS_1 ; TS_2$. There exists s'', h'' and s'_1, h'_1 such that $s, h \llbracket TS_1 \rrbracket s'', h''$ and $s'', h'' \llbracket TS_2 \rrbracket s', h'$ and $s_1, h_1 \llbracket TS_1 \rrbracket s'_1, h'_1$ and $s'_1, h'_1 \llbracket TS_2 \rrbracket s'_1, h'_1$. We assume $s \& s_1 \models_{\beta} \Theta$ and $h \& h_1 \models_{\beta} \mathcal{I}$, and also assume that there exists $\theta'_0 \in \Theta'$ such that $s' \models \text{ant}(\theta'_0)$ and $s'_1 \models \text{ant}(\theta'_0)$; by Lemma 4.6 we infer that there exists $\theta''_0 \in \Theta''$ such that $s'' \models \text{ant}(\theta''_0)$ and $s'_1 \models \text{ant}(\theta''_0)$. Therefore we can apply the induction hypothesis to TS_1 to find β'' extending β such that $s'' \& s'_1 \models_{\beta''} \Theta''$ and $h'' \& h'_1 \models_{\beta''} \mathcal{I}$, and next apply the induction hypothesis to TS_2 to find β' extending β'' such that $s' \& s'_1 \models_{\beta'} \Theta'$ and $h' \& h'_1 \models_{\beta'} \mathcal{I}$. This is as desired, since β' extends β .

$TS = x := A$. We shall prove the claim with $\beta' = \beta$, and (since $h' = h$ and $h'_1 = h_1$) the only non-trivial part is $s' \& s'_1 \models_{\beta'} \Theta'$. Here $s' = [s \mid x \mapsto v]$ with $v = \llbracket A \rrbracket^s$, and $s'_1 = [s_1 \mid x \mapsto v_1]$ with $v_1 = \llbracket A \rrbracket^{s_1}$. Let $(\phi' \Rightarrow E' \times) \in \Theta'$ be given, and assume that $s' \models \phi'$ and $s'_1 \models \phi'$; we must prove $\llbracket E' \rrbracket^{s'} \beta \llbracket E' \rrbracket^{s'_1}$. With $\phi = \phi'[A/x]$ and $E = E'[A/x]$ we have $\phi \Rightarrow E \times \in \Theta$ and thus $s \& s_1 \models_{\beta} \phi \Rightarrow E \times$. Since $s \models \phi$ and $s_1 \models \phi$ follow from $s' \models \phi'$ and $s'_1 \models \phi'$ by Lemma A.7, we infer $\llbracket E \rrbracket^s \beta \llbracket E \rrbracket^{s_1}$. But this amounts to the desired $\llbracket E' \rrbracket^{s'} \beta \llbracket E' \rrbracket^{s'_1}$, since by Lemma A.5 we have $\llbracket E' \rrbracket^{s'} = \llbracket E \rrbracket^s$ and $\llbracket E' \rrbracket^{s'_1} = \llbracket E \rrbracket^{s_1}$.

$TS = \text{if } B \text{ then } TS_1 \text{ else } TS_2$. Except for symmetry, there are two cases. The first case is when $\llbracket B \rrbracket^s = \llbracket B \rrbracket^{s_1} = \text{True}$, in which case we have $s, h \llbracket TS_1 \rrbracket s', h'$ and $s_1, h_1 \llbracket TS_1 \rrbracket s'_1, h'_1$. It is sufficient to establish $s \& s_1 \models_{\beta} \Theta_1$, since then induction on TS_1 will yield β' extending β such that $s' \& s'_1 \models_{\beta'} \Theta'$ and $h' \& h'_1 \models_{\beta'} \mathcal{I}$. So given $(\phi_1 \Rightarrow E_1 \times) \in \Theta_1$, and assuming $s \models \phi_1$ and $s_1 \models \phi_1$, our obligation is to show $\llbracket E_1 \rrbracket^s \beta \llbracket E_1 \rrbracket^{s_1}$. By Lemma 4.3 (Totality) applied to TS_1 , there exists $\theta' \in \Theta'$ such that $(\phi_1 \Rightarrow E_1 \times, \neg, \theta') \in R_1$. Two cases:

- if $\theta' \in \Theta'_m$ then, by R'_1 , $(\phi_1 \wedge B \Rightarrow E_1 \times) \in \Theta$.
- if $\theta' \in \Theta'_u$ then, by R_0 and Lemma 4.3, there exists ϕ_2 such that $(\phi_1 \wedge B) \vee (\phi_2 \wedge \neg B) \Rightarrow E_1 \times \in \Theta$.

Since $s \models \phi_1 \wedge B$ and $s_1 \models \phi_1 \wedge B$, and since $s \& s_1 \models_{\beta} \Theta$, in both cases we can infer the desired $\llbracket E_1 \rrbracket^s \beta \llbracket E_1 \rrbracket^{s_1}$.

The second case to be considered is when $\llbracket B \rrbracket^s = \text{False}$ but $\llbracket B \rrbracket^{s_1} = \text{True}$, in which case we have $s, h \llbracket TS_2 \rrbracket s', h'$ and $s_1, h_1 \llbracket TS_1 \rrbracket s'_1, h'_1$. We shall prove the claim with $\beta' = \beta$, that is, show $s' \& s'_1 \models_{\beta} \Theta'$ and $h' \& h'_1 \models_{\beta} \mathcal{I}$. First we shall show

$$\forall \theta' \in \Theta' : (s' \models \text{ant}(\theta') \wedge s'_1 \models \text{ant}(\theta')) \Rightarrow \theta' \in \Theta'_u \quad (2)$$

by the following argument: if $\theta' \in \Theta'$ with $s' \models \text{ant}(\theta')$ and $s'_1 \models \text{ant}(\theta')$, by Lemma 4.6 there exists $(\phi_1 \Rightarrow E_1 \times, \neg, \theta') \in R_1$ and $(\phi_2 \Rightarrow E_2 \times, \neg, \theta') \in R_2$ such that $s \models \phi_2$ and $s_1 \models \phi_1$. Assume, to get a contradiction, that $\theta' \in \Theta'_m$; then by R'_0 we would have $\phi \Rightarrow B \times \in \Theta$ with $\phi = (\phi_1 \wedge B) \vee (\phi_2 \wedge \neg B)$,

where $s \models \phi$ (since $s \models \phi_2 \wedge \neg B$) and $s_1 \models \phi$ (since $s_1 \models \phi_1 \wedge B$). But then from $s \& s_1 \models_{\beta} \Theta$ and hence $s \& s_1 \models_{\beta} \phi \Rightarrow B \times$ we would have $\llbracket B \rrbracket^s \beta \llbracket B \rrbracket^{s_1}$, contradicting $\llbracket B \rrbracket^s \neq \llbracket B \rrbracket^{s_1}$.

Having established (2), we shall show $s' \& s'_1 \models_{\beta} \Theta'$. Let $\theta' = (\phi' \Rightarrow E \times) \in \Theta'$, and assume $s' \models \phi'$ and $s'_1 \models \phi'$; we must prove $\llbracket E \rrbracket^{s'} \beta \llbracket E \rrbracket^{s'_1}$. By (2) we see that $\theta' \in \Theta'_u$ and then infer by Lemma 4.3 that there exists unique $\theta \in \Theta$ such that $(\theta, _, \theta') \in R$ and also that $E = \text{con}(\theta)$. By Lemma 4.6, with $\phi = \text{ant}(\theta)$, we infer that $s \models \phi$ and $s_1 \models \phi$. From $s \& s_1 \models_{\beta} \Theta$ we thus infer $\llbracket E \rrbracket^s \beta \llbracket E \rrbracket^{s_1}$. But since Lemma 4.3 also states that s, s' agree on $\text{fv}(E)$ and that s_1, s'_1 agree on $\text{fv}(E)$, this amounts to the desired $\llbracket E \rrbracket^{s'} \beta \llbracket E \rrbracket^{s'_1}$.

Finally, we shall show $h' \& h'_1 \models_{\beta} \mathcal{I}$, and therefore consider l, l_1 with $l \beta l_1$ (thus $l \in \text{dom}(h)$ and $l_1 \in \text{dom}(h_1)$). With $r = h(l)$ and $r_1 = h_1(l_1)$ and $r' = h'(l)$ and $r'_1 = h'_1(l_1)$, we assume $r \& r_1 \models_{\beta} \mathcal{I}$ and must prove $r' \& r'_1 \models_{\beta} \mathcal{I}$. Let $\phi_0 \Rightarrow E_0 \times \in \mathcal{I}$ be given, and assume $r' \models \phi_0$ and $r'_1 \models \phi_0$; we must prove $\llbracket E_0 \rrbracket_{r'} \beta \llbracket E_0 \rrbracket_{r'_1}$. By our overall assumption, there exists $\theta'_0 \in \Theta'$ with $s' \models \text{ant}(\theta'_0)$ and $s'_1 \models \text{ant}(\theta'_0)$; by (2) we infer that $\theta'_0 \in \Theta'_u$. Thus R contains no $(_, m, \theta'_0)$, so by Lemma 4.7 applied to TS we infer first $r \models \phi_0$ and $r_1 \models \phi_0$, which since $r \& r_1 \models_{\beta} \phi_0 \Rightarrow E_0 \times$ implies $\llbracket E_0 \rrbracket_r \beta \llbracket E_0 \rrbracket_{r_1}$, and next that for all $f \in \text{ff}(E_0)$ we have $r(f) = r'(f)$ and $r_1(f) = r'_1(f)$, yielding the desired $\llbracket E_0 \rrbracket_{r'} \beta \llbracket E_0 \rrbracket_{r'_1}$.

$TS = \text{new } x \text{ in } RS \text{ close}$. We assume $s, h \llbracket S \rrbracket s', h'$ and $s_1, h_1 \llbracket S \rrbracket s'_1, h'_1$, that is there exists $l \notin \text{dom}(h) \cup \text{ran}(h) \cup \text{ran}(s)$ and $l_1 \notin \text{dom}(h_1) \cup \text{ran}(h_1) \cup \text{ran}(s_1)$ such that $[s \mid x \mapsto l], \text{deflt} \llbracket RS \rrbracket s', r'$ and $[s_1 \mid x \mapsto l_1], \text{deflt} \llbracket RS \rrbracket s'_1, r'_1$ where $h' = [h \mid l \mapsto r']$ and $h'_1 = [h_1 \mid l_1 \mapsto r'_1]$.

We now define β' as $\beta \cup \{(l, l_1)\}$. It is sufficient to establish

$$[s \mid x \mapsto l], \text{deflt} \& [s_1 \mid x \mapsto l_1], \text{deflt} \models_{\beta'} \Theta_0 \quad (3)$$

as then Prop. 4.1 tells us that $s', r' \& s'_1, r'_1 \models_{\beta'} \Theta' \cup \mathcal{I}$, in particular $s' \& s'_1 \models_{\beta'} \Theta'$ and $h'(l) \& h'_1(l_1) \models_{\beta'} \mathcal{I}$. (To establish $h' \& h'_1 \models_{\beta'} \mathcal{I}$, we must also prove that $h'(l') \& h'_1(l'_1) \models_{\beta'} \mathcal{I}$ when $l' \beta l'_1$. But then $h'(l') = h(l')$ and $h'_1(l'_1) = h_1(l'_1)$, so the result follows from $h \& h_1 \models_{\beta} \mathcal{I}$.)

We now embark on proving (3), so let $\theta_0 \in \Theta_0$ with $\theta_0 = (\phi_0 \Rightarrow E_0 \times)$, and assume $[s \mid x \mapsto l], \text{deflt} \models \phi_0$ and $[s_1 \mid x \mapsto l_1], \text{deflt} \models \phi_0$; we must prove $\llbracket E_0 \rrbracket_{\text{deflt}}^{[s \mid x \mapsto l]} \beta' \llbracket E_0 \rrbracket_{\text{deflt}}^{[s_1 \mid x \mapsto l_1]}$. By repeated application of Lemma A.8, we infer $[s \mid x \mapsto l] \models \phi_0 \overline{\text{deflt}(f)/f}$ and $[s_1 \mid x \mapsto l_1] \models \phi_0 \overline{\text{deflt}(f)/f}$, and by Lemma A.9 we next infer $s \models \phi$ and $s_1 \models \phi$ where $\phi = \text{rm}_x^{\ell}(\phi_0 \overline{\text{deflt}(f)/f})$. Note that with $E = \text{rm}_x^{\ell}(E_0 \overline{\text{deflt}(f)/f})$ we have $(\phi \Rightarrow E \times) \in \Theta$, so from $s \& s_1 \models_{\beta} \Theta$ we conclude that $\llbracket E \rrbracket^s \beta \llbracket E \rrbracket^{s_1}$. We now first consider the case when $x \notin \text{fv}(E_0)$; here $E = E_0 \overline{\text{deflt}(f)/f}$ so by repeated application of Lemma A.6 we infer $\llbracket E_0 \rrbracket_{\text{deflt}}^s \beta \llbracket E_0 \rrbracket_{\text{deflt}}^{s_1}$ and thus the desired $\llbracket E_0 \rrbracket_{\text{deflt}}^{[s \mid x \mapsto l]} \beta' \llbracket E_0 \rrbracket_{\text{deflt}}^{[s_1 \mid x \mapsto l_1]}$. In the case when $x \in \text{fv}(E_0)$, that result follows directly from Lemma A.10.

$TS = \text{open } x \text{ in } RS \text{ close}$. We assume $s, h \llbracket TS \rrbracket s', h'$ and $s_1, h_1 \llbracket TS \rrbracket s'_1, h'_1$, that is with $l = s(x)$ and $r = h(l)$ and $l_1 = s_1(x)$ and $r_1 = h_1(l_1)$ we have $s, r \llbracket RS \rrbracket s', r'$ and $s_1, r_1 \llbracket RS \rrbracket s'_1, r'_1$ where $h' = [h \mid l \mapsto r']$ and $h'_1 = [h_1 \mid l_1 \mapsto r'_1]$. We shall prove the claim with $\beta' = \beta$, that is we show $s' \& s'_1 \models_{\beta} \Theta'$ and $h' \& h'_1 \models_{\beta} \mathcal{I}$, given $s \& s_1 \models_{\beta} \Theta$ and $h \& h_1 \models_{\beta} \mathcal{I}$.

We shall first consider the case when $l \beta l_1$, where $r \& r_1 \models_{\beta} \mathcal{I}$ and it is sufficient to show

$$s, r \& s_1, r_1 \models_{\beta} \Theta_0 \quad (4)$$

as then Prop. 4.1 tells us that $s', r' \& s'_1, r'_1 \models_{\beta} \Theta' \cup \mathcal{I}$, in particular $s' \& s'_1 \models_{\beta} \Theta'$ and $h'(l) \& h'_1(l_1) \models_{\beta} \mathcal{I}$. (To establish $h' \& h'_1 \models_{\beta} \mathcal{I}$, we must also prove that $h'(l') \& h'_1(l'_1) \models_{\beta} \mathcal{I}$ when $l' \beta l'_1$ but $l' \neq l, l'_1 \neq l_1$. But then $h'(l') = h(l')$ and $h'_1(l'_1) = h_1(l'_1)$, and the result follows from $h \& h_1 \models_{\beta} \mathcal{I}$.)

We now embark on proving (4), so let $\theta_0 \in \Theta_0$ with $\theta_0 = (\phi_0 \Rightarrow E_0 \times)$ so as to prove $s, r \& s_1, r_1 \models_{\beta} \theta_0$. Since by Fact A.11, ϕ_0 logically implies $\text{ff}^+(\phi_0)$, it will be sufficient to prove that $s, r \& s_1, r_1 \models_{\beta}$

$ff^+(\phi_0) \Rightarrow E_0 \times$. By Lemma 4.3 applied to RS , there exists $\theta' \in \Theta' \cup \mathcal{I}$ such that $(\theta_0, \gamma, \theta') \in R_0$. There are now four cases:

$\gamma = m$, E_0 **is field-free**. From R_3 we see that Θ contains $ff^+(\phi_0) \Rightarrow E_0 \times$, and the claim follows from $s \& s_1 \models_{\beta} \Theta$.

$\gamma = m$, E_0 **is not field-free**. Since $(\mathcal{I} \triangleright^2 \theta_0) \in VC_2 \subseteq VC$, the claim follows from $\models VC$ and $r \& r_1 \models_{\beta} \mathcal{I}$.

$\gamma = u$, $\theta' \in \Theta'$. From R_4 we see that Θ contains $ff^+(\phi_0) \Rightarrow E_0 \times$, and the claim follows from $s \& s_1 \models_{\beta} \Theta$.

$\gamma = u$, $\theta' \in \mathcal{I}$. Let $\theta' = (\phi' \Rightarrow E' \times)$. Since $(\phi_0 \triangleright^1 \phi') \in VC_1 \subseteq VC$, from $\models VC$ we infer that ϕ_0 logically implies ϕ' , and from Lemma 4.3 we get $E' = E_0$. Thus θ' logically implies θ_0 , and the claim follows from $r \& r_1 \models_{\beta} \mathcal{I}$.

We shall now consider the case when $l \beta l_1$ does not hold, and first show $s' \& s'_1 \models_{\beta} \Theta'$. Let $\theta' \in \Theta'$ be given with $\theta' = (\phi' \Rightarrow E' \times)$, and assume $s' \models \phi'$ and $s'_1 \models \phi'$; we must prove $\llbracket E' \rrbracket^{s'} \beta \llbracket E' \rrbracket^{s'_1}$. By Lemma 4.6 applied to RS , there exists $\theta = (\phi \Rightarrow E \times)$ such that $(\theta, \gamma, \theta') \in R_0$ and such that $s, r \models \phi$ and $s_1, r_1 \models \phi$, which (by Fact A.11) implies $s \models ff^+(\phi)$ and $s_1 \models ff^+(\phi)$. Here it holds that $\gamma = u$. For assume otherwise, that is $\gamma = m$, then from R_1 we see that $(ff^+(\phi) \Rightarrow x \times) \in \Theta$, so from $s \& s_1 \models_{\beta} \Theta$ we infer $s(x) \beta s_1(x)$, that is $l \beta l_1$, contradicting our case assumption. Thus $\gamma = u$ and we can apply Lemma 4.3 to RS to infer that $E' = E$, and that s, s' agree on $\text{fv}(E)$, and that s_1, s'_1 agree on $\text{fv}(E)$. From R_4 we see that $ff^+(\phi) \Rightarrow E \times \in \Theta$, so from $s \& s_1 \models_{\beta} \Theta$ we infer $\llbracket E \rrbracket^s \beta \llbracket E \rrbracket^{s_1}$ which amounts to the desired $\llbracket E' \rrbracket^{s'} \beta \llbracket E' \rrbracket^{s'_1}$.

We are left with showing $h' \& h'_1 \models_{\beta} \mathcal{I}$, and thus consider l', l'_1 with $l' \beta l'_1$; assuming $h(l') \& h_1(l'_1) \models_{\beta} \mathcal{I}$, we must prove $h'(l') \& h'_1(l'_1) \models_{\beta} \mathcal{I}$. This is obvious if $l' \neq l$ and $l'_1 \neq l_1$ since then $h'(l') = h(l')$ and $h'_1(l'_1) = h_1(l'_1)$. We therefore assume, wlog., that $l' = l$, in which case $l'_1 \neq l_1$ (since the bijection β relates l' to l'_1 but does not relate l to l_1) and thus there exists r_0 with $r_0 = h_1(l'_1) = h'_1(l'_1)$. (Recall that we defined $r = h(l)$ and $r' = h'(l)$.) Assuming $r \& r_0 \models_{\beta} \mathcal{I}$, we must prove $r' \& r_0 \models_{\beta} \mathcal{I}$. Let $\theta_0 \in \mathcal{I}$ be given with $\theta_0 = (\phi_0 \Rightarrow E_0 \times)$, and assume $r' \models \phi_0$ and $r_0 \models \phi_0$; we must prove $\llbracket E_0 \rrbracket_{r'} \beta \llbracket E_0 \rrbracket_{r_0}$. By Lemma 4.3 applied to RS , there exists $(\theta, \gamma, \theta_0) \in R_0$. Here it cannot be the case that $\gamma = m$ for then R_2 would cause Θ to contain $(\text{true} \Rightarrow x \times)$ which (since $s \& s_1 \models_{\beta} \Theta$) implies $s(x) \beta s_1(x)$, that is $l \beta l_1$, contradicting our case assumption. Thus $\gamma = u$ and we can further apply Lemma 4.3 to RS to infer that r', r agree on $\text{fv}(E_0)$; we also infer that θ is unique with $(\theta, _, \theta_0) \in R_0$ so Lemma 4.6, applied to RS , tells us that $s, r \models \text{ant}(\theta)$. Since $(\text{ant}(\theta) \triangleright^1 \phi_0) \in VC_1 \subseteq VC$, from $\models VC$ we infer that $r \models \phi_0$. Together with $r_0 \models \phi_0$ and $r \& r_0 \models_{\beta} \mathcal{I}$, this implies $\llbracket E_0 \rrbracket_r \beta \llbracket E_0 \rrbracket_{r_0}$ and hence the desired $\llbracket E_0 \rrbracket_{r'} \beta \llbracket E_0 \rrbracket_{r_0}$.

$TS = \text{while } B \text{ do } TS_0$. It is sufficient to prove the following result: assume $s \& s_1 \models_{\beta} \Theta$ and $h \& h_1 \models_{\beta} \mathcal{I}$, and assume that (with f_i as defined in Fig. 4) there exists i, j such that $s, h \models f_i s', h'$ and $s_1, h_1 \models f_j s'_1, h'_1$; further assume that there exists $\theta' \in \Theta$ such that $s' \models \text{ant}(\theta')$ and $s'_1 \models \text{ant}(\theta')$. Then there exists β' extending β such that $s' \& s'_1 \models_{\beta'} \Theta$ and $h' \& h'_1 \models_{\beta'} \mathcal{I}$.

We shall prove this result by induction in $i + j$. If $i = j = 0$, the claim is obvious, with $\beta' = \beta$, as then $s' = s, h' = h, s'_1 = s_1, h'_1 = h_1$. Apart from symmetry, there are two other cases.

First assume $i > 0, j > 0$. Thus $\llbracket B \rrbracket^s = \text{True} = \llbracket B \rrbracket^{s_1}$, and there exists s'', h'', s'_1, h'_1 such that $s, h \models TS_0 s'', h''$ and $s'', h'' \models f_{i-1} s', h'$ and $s_1, h_1 \models TS_0 s'_1, h'_1$ and $s'_1, h'_1 \models f_{j-1} s'_1, h'_1$. We shall now establish

$$s \& s_1 \models_{\beta} \Theta \tag{5}$$

and therefore assume that $(\phi_0 \Rightarrow E_0 \times) \in \Theta_0$, and that $s \models \phi_0$ and $s_1 \models \phi_0$; we must show $\llbracket E_0 \rrbracket^s \beta \llbracket E_0 \rrbracket^{s_1}$. Since $\Theta \triangleright^2 (\phi_0 \wedge B \Rightarrow E_0 \times) \in VC_1 \subseteq VC$, we from $\models VC$ infer $s \& s_1 \models_{\beta} \phi_0 \wedge B \Rightarrow E_0 \times$, and the claim follows as $s \models \phi_0 \wedge B$ and $s_1 \models \phi_0 \wedge B$.

We return to our main proof obligation where Lemma 4.6, applied to TS , shows (since $s'', h'' \llbracket TS \rrbracket s', h'$ and $s''_1, h''_1 \llbracket TS \rrbracket s'_1, h'_1$) that there exists $\theta'' \in \Theta$ such that $s'' \models \text{ant}(\theta'')$ and $s''_1 \models \text{ant}(\theta'')$. Therefore we can apply the outermost induction hypothesis to TS_0 , and from (5) find β'' extending β such that $s'' \& s''_1 \models_{\beta''} \Theta$ and $h'' \& h''_1 \models_{\beta''} \mathcal{I}$. We next apply the innermost induction hypothesis to find β' extending β'' such that $s' \& s'_1 \models_{\beta'} \Theta$ and $h' \& h'_1 \models_{\beta'} \mathcal{I}$. This is as desired, since β' extends β .

Next assume $i > 0, j = 0$. Then $\llbracket B \rrbracket^s = \text{True}$ and $\llbracket B \rrbracket^{s_1} = \text{False}$, so $s'_1 = s_1$ and $h'_1 = h_1$. We shall prove the claim with $\beta' = \beta$. First we establish

$$\text{if } \theta \in \Theta \text{ and } s' \models \text{ant}(\theta), s_1 \models \text{ant}(\theta) \text{ then } \theta \in \Theta_u \quad (6)$$

by contradiction: if $\theta \in \Theta_m$ then Lemma 4.6, applied to TS , tells us that $s \models \phi_m$. Since $(\text{ant}(\theta) \triangleright^1 \phi_m) \in VC_3 \subseteq VC$, from $\models VC$ we infer that $s_1 \models \phi_m$. Since $(\Theta \triangleright^2 \phi_m \Rightarrow B \times) \in VC_2 \subseteq VC$, from $\models VC$ and $s \& s_1 \models_{\beta} \Theta$ we infer that $s \& s_1 \models_{\beta} \phi_m \Rightarrow B \times$, and thus $\llbracket B \rrbracket^s \beta \llbracket B \rrbracket^{s_1}$ which is a contradiction as $\text{True} \beta \text{False}$ cannot hold. We have thus established (6).

We return to our main proof obligations, and first show $s' \& s_1 \models_{\beta} \Theta$. So consider $\theta \in \Theta$ with $\theta = (\phi \Rightarrow E \times)$, and assume that $s' \models \phi$ and $s_1 \models \phi$; we must show $\llbracket E \rrbracket^{s'} \beta \llbracket E \rrbracket^{s_1}$. From (6) we infer that $\theta \in \Theta_u$ and therefore Lemma 4.6, applied to TS , tells us that $s \models \phi$. From $s \& s_1 \models_{\beta} \theta$ we thus infer $\llbracket E \rrbracket^s \beta \llbracket E \rrbracket^{s_1}$; this is as desired since Lemma 4.3, applied to TS , tells us that s, s' agree on $\text{fv}(E)$.

Finally, we shall show $h \& h_1 \models_{\beta} \mathcal{I}$. Thus consider l, l_1 with $l \beta l_1$; with $r = h(l)$ and $r' = h'(l)$ and $r_1 = h_1(l_1)$ we must prove $r' \& r_1 \models_{\beta} \mathcal{I}$, where we can assume $r \& r_1 \models_{\beta} \mathcal{I}$. Thus consider $\theta_0 \in \mathcal{I}$ with $\theta_0 = (\phi_0 \Rightarrow E_0 \times)$, and assume $r' \models \phi_0$ and $r_1 \models \phi_0$; we must prove $\llbracket E_0 \rrbracket_{r'} \beta \llbracket E_0 \rrbracket_{r_1}$. Recall that one of the inductive assumptions is that θ' is such that $s' \models \text{ant}(\theta')$ and $s'_1 \models \text{ant}(\theta')$, so from (6) we infer that $\theta' \in \Theta_u$, implying that R contains no $(-, m, \theta')$. We can therefore apply Lemma 4.7 to TS (as $s, h \llbracket TS \rrbracket s', h'$) and infer that $r \models \phi_0$, which together with $r \& r_1 \models_{\beta} \mathcal{I}$ implies $\llbracket E_0 \rrbracket_r \beta \llbracket E_0 \rrbracket_{r_1}$; this is as desired since Lemma 4.7 also tells us that for all $f \in \text{ff}(E_0)$, $r(f) = r'(f)$. \square