Verification Condition Generation for Conditional Information Flow*
DRAFT as of June 17, 2007

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

Anindya Banerjee
IBM T. J. Watson Research Center
Hawthorne, NY, USA
ab@cis.ksu.edu

June 17, 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 secrets/classified and L for public/observable/unclassified. For a given policy, a program P satisfies *noninterference* (NI) [17] 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; those can be automatically checked by tools such as BLAST [19], 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.

Given a variable x labeled L, the formulation of noninterference entails that we restrict our attention to pair 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 $x\ltimes$, where two states σ_1, σ_2 satisfy $x\ltimes$ when $\sigma_1(x) = \sigma_2(x)$. If a program P has observable input variables $x_1, \ldots, x_n$, and observable output variables $y_1, \ldots, y_m$, then NI can be recast as

$$\{x_1\ltimes \wedge \ldots \wedge x_n\ltimes\}\ P\ \{y_1\ltimes \wedge \ldots \wedge y_m\ltimes\}$$

The meaning (partial correctness) of the above triple is that for any two states σ_1, σ_2 that agree on the values of $x_1, \ldots, x_n$ (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 $y_1, \ldots, y_m$ 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 information flow analysis for sequential, heap-manipulating programs. If a triple is derivable for a program then NI holds for the program. The specification is flow sensitive (unlike most type-based approaches), can check information leaks caused by aliasing, and can be used for analyzing observational purity. Moreover, the specification can be used to check compliance with delimited release policies [22] 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].

The logical specification of [1] comes with an analysis algorithm which, however, has some shortcomings: it needs to know the shape of the heap, and it does not integrate well with programmer assertions. Also, the specification itself does not 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 [24, 4, 20]).

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. We prove the correctness of the algorithm, and use it to verify some naturally occurring examples. A prototype implementation[2] is currently being developed by Jonathan Hoag.

An example loop flow invariant is $x\ltimes$, 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 $i > n \Rightarrow x\ltimes$, 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

[1] Two remarks: (a) The connection with NI based on security labels [24] is that for any well-labeled program, P, if $l_1, \ldots, l_n$ are all the L-variables in P then $l_1\ltimes \wedge \ldots \wedge l_n\ltimes$ 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".

[2] Available 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.

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

$$\mathbf{if}\ w\ \mathbf{then}\ x := 7\ \mathbf{else}\ x := 7; \mathbf{assert}(x = 7)$$

Given the postcondition $x\ltimes$, the algorithm will compute $x = 7 \Rightarrow x\ltimes$ 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 $x = 7 \Rightarrow x\ltimes$ 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 $w\ltimes$. 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.

Loop flow invariants. Consider the program P in Fig. 1(a), and the policy specification $\{x\ltimes\}\ \_\ \{result\ltimes\}$. 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 $result$ agree? Note that the precondition does not make any commitments about $v\ltimes$ and $h\ltimes$.

To answer the above question, observe that since the program updates $result$ (line 4), for $result\ltimes$ to hold at the end, $v\ltimes$ must also hold. Alas, $v\ltimes$ holds only at the beginning of every *odd* iteration of the loop — but fortunately, this is exactly when v is used to update $result$. It turns out that to verify the program we need the loop flow invariant $odd(i) \Rightarrow v\ltimes$ which testifies to *conditionally secure information flow* within the loop.[3] Furthermore, after $result$ is updated, the assignment to v (line 5) *invalidates* the invariant because $h\ltimes$ need 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 the above loop flow invariant, generates valid verification conditions (VCs) together with a precondition that includes $x\ltimes$ but *not* $h\ltimes$. 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 $result$ has type L and h has type H, reject the program as insecure. (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.) For, well-typedness demands v to have type L, due to the assignment to $result$ (line 4), and also to have type H, due to the assignment to v (line 5).

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

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

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

(a)

```
open x in
    y := .src;
    i := .idx;
close;
open y in
    assert (odd(i) → odd(.idx));
    q := .val;
close;
open x in
    assert (.idx = i);
    .val := q;
    result := .val;
close;
```

(b)

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.

"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

$$odd(o.idx) \Rightarrow (o.val)\ltimes \text{ and}$$
$$odd(o.idx) \Rightarrow (o.src)\ltimes.$$

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

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

satisfies the policy $\{x\ltimes\}\ \_\ \{odd(i) \Rightarrow result\ltimes\}$.

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:

$$y := x.src;\ i := x.idx;$$
$$\textbf{assert}\ (odd(i) \rightarrow odd(y.idx));$$
$$q := y.val;\ x.val := q;\ result := x.val$$

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 $\textbf{open}\ x\ \textbf{in}\ y := .f;\ \textbf{close}$. A field update, $x.f := y$, occurs as $\textbf{open}\ x\ \textbf{in}\ .f := y;\ \textbf{close}$.

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 $\mathbf{assert}(odd(x.idx) \rightarrow odd(y.idx))$, we need to save the result of $x.idx$ into a variable i. Finally, 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) can 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) with postcondition $odd(i) \Rightarrow result\ltimes$ and object flow invariant $\{odd(.idx) \Rightarrow .val\ltimes,\ odd(.idx) \Rightarrow .src\ltimes\}$, generates (as sketched in Sect. 5) valid VCs, and the precondition $true \Rightarrow x\ltimes$ (equivalent to $x\ltimes$).

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 $.val < 10 \Rightarrow val\ltimes$.

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 $result$ is taken from the list pointed to by y, the overall policy specification is $\{x\ltimes\}\ \_\ \{result\ltimes\}$. Our algorithm verifies that the program in Fig. 2 satisfies this specification, in that from postcondition $result\ltimes$ it generates precondition $x\ltimes$ 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.

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

Here $(.val < 10)\ltimes$ is satisfies 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 a 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 $\mathbf{new} \ldots \mathbf{close}$ is exited the object is in a "steady state".

Readers familiar with the Boogie methodology [6] might notice some similarity between $\mathbf{open} \ldots \mathbf{close}$ and Boogie's $\mathbf{unpack}$ and $\mathbf{pack}$, 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 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.

```
y := nil; z := nil;
while x ≠ nil do
  open x in assert(.t = 1); v := .val; n := .next; close;
  x := n;
  if v < 10
  then new y1 in .val := v; .t := 2; .next := y; close;
      y := y1;
  else new z1 in .val := v; .t := 3; .next := z; close;
      z := z1;
  fi;
od;
```

```
result := nil;
while y ≠ nil do
  open y in
    assert(.t = 2);
    result := .val;
    y := .next;
  close;
od
```

Figure 2: List splitting

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

$$\begin{array}{lcl} A & ::= & x \mid .f \mid c \mid \mathsf{nil} \mid A \,\mathsf{op}\, A \\ B & ::= & A \,\mathsf{bop}\, A \end{array}$$

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

We write $\mathrm{fv}(E)$ (or $\mathrm{ff}(E)$) 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 $\{\mathsf{int}, \mathsf{obj}\}$. 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 $type(x) = \mathsf{obj}$. Then $x \in \mathrm{fv}(A)$ implies $A = x$, and $x \in \mathrm{fv}(B)$ implies that B is either $x = x$ or $A = x$ or $x = A$ with $x \notin \mathrm{fv}(A)$.*

Semantic domains. A value ($v \in Val$) is an integer n, a location $l \in Loc$, or nil; default values are defined as $\mathit{deflt}(\mathsf{int}) = 0$ and $\mathit{deflt}(\mathsf{obj}) = nil$, and we write $\mathit{deflt}(f)$ for $\mathit{deflt}(type(f))$. A *store* $s \in Store$ maps variables to values, an *object state* r maps field names to values, and a *heap* $h \in Heap$ maps locations to object states; the notions of $dom(\_)$ and $ran(\_)$ are as usual except that (with misuse of notation) we write $ran(h) = \{v \mid \exists l \in dom(h), f \in \mathbf{Fld} \bullet v = h(l)(f)\}$. We write $[s \mid x \mapsto v]$ for the store that is like s except that it maps x into v; similarly we write $[r \mid f \mapsto v]$ and $[h \mid l \mapsto r]$.

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.

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

Figure 3: Command syntax

$[\![x]\!]_r^s = s(x)$, $\quad [\![.f]\!]_r^s = r(f)$, $\quad [\![c]\!]_r^s = c$, $\quad [\![\mathsf{nil}]\!]_r^s = nil$
$[\![A_1 + A_2]\!]_r^s = [\![A_1]\!]_r^s + [\![A_2]\!]_r^s$, etc.
$[\![A_1 < A_2]\!]_r^s = \mathsf{True}$ iff $[\![A_1]\!]_r^s < [\![A_2]\!]_r^s$, etc.

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

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

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 by the store s and the object state r. The definition is inductive in ϕ: $s,r \models B$ iff $[\![B]\!]_r^s = \mathsf{True}$; $s,r \models \phi_1 \wedge \phi_2$ 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 $s,r\ [\![RS]\!]\ s',r'$. A top-level command transforms a store and a heap into another store and another heap; thus its semantics is given in the form $s,h\ [\![TS]\!]\ s',h'$. The semantics is defined inductively on RS and TS; some key clauses are given in Fig. 4. Note that for some TS and s, h, there may not exist any s', h' such that $s,h\ [\![TS]\!]\ s',h'$ (modulo the choice of fresh location for object allocation, there exists at most one s', h'); this can happen if a $\mathbf{while}$ loop does not terminate, or an $\mathbf{assert}$ fails.

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 $\phi \Rightarrow E\ltimes$ which intuitively is satisfied by a pair of states if either

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

Figure 4: Command semantics, selected clauses

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, $nil\ \beta\ nil$, True β True, and False β False.

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

$s,r\&s_1,r_1 \models_\beta \phi \Rightarrow E\ltimes$ iff whenever $s,r \models \phi$ and $s_1,r_1 \models \phi$ then $[\![E]\!]^s_r\ \beta\ [\![E]\!]^{s_1}_{r_1}$.

For $\theta = (\phi \Rightarrow E\ltimes)$, we call ϕ the antecedent of θ and write $\phi = ant(\theta)$, and we call E the consequent of θ and write $E = con(\theta)$. 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 use $\Theta \in \mathcal{P}(\mathbf{2Assert})$ to range over sets of 2-assertions, with conjunction implicit. Thus

$s,r\&s_1,r_1 \models_\beta \Theta$ iff $\forall \theta \in \Theta \bullet s,r\&s_1,r_1 \models_\beta \theta$.

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

$$\{pin = 1234 \Rightarrow out\ltimes, pin \neq 1234 \Rightarrow out\ltimes\}$$

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 $(pin = 1234 \vee pin \neq 1234) \Rightarrow out\ltimes$ (which is just $true \Rightarrow out\ltimes$).

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

$h\&h_1 \models_\beta \mathcal{I}$ iff for all l, l_1 with $l\ \beta\ l_1$:
$h(l)\&h_1(l_1) \models_\beta \mathcal{I}$.

4 Algorithm

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

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

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 $dom(R) = \{\theta \mid \exists(\theta, \_, \_) \in R\}$ and $ran(R) = \{\theta' \mid \exists(\_, \_, \theta') \in R\}$. 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 $(q > 4 \Rightarrow w\ltimes, m, q > 4 \Rightarrow x\ltimes)$ and $(w > 3 \Rightarrow z\ltimes, u, x > 3 \Rightarrow z\ltimes)$.

Verification conditions. These are either of the form $\phi \rhd^1 \phi'$, meaning that ϕ logically implies ϕ', or of the form $\Theta \rhd^2 \theta$, again meaning that Θ logically implies θ but now for 2-assertions. Thus $\models \phi \rhd^1 \phi'$ iff for all s, r: whenever $s,r \models \phi$ then also $s,r \models \phi'$; and $\models \Theta \rhd^2 \theta$ 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.

Correctness results. Ultimately, we must express that if $[VC]\{\Theta\}\ (\_) \Longleftarrow S\ \{\Theta'\}$ 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. $[VC]\{\Theta\}\ (\_) \Longleftarrow RS\ \{\Theta'\}$ *and that* $\models VC$
2. $s,r\ [\![RS]\!]\ s',r'$ *and* $s_1,r_1\ [\![RS]\!]\ s_1',r_1'$
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-*in*sensitive, 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. $[VC]\{\Theta\}\ (\_) \Longleftarrow TS\ \{\Theta'\}$ *and that* $\models VC$
2. $s,h\ [\![TS]\!]\ s',h'$ *and that* $s_1,h_1\ [\![TS]\!]\ s_1',h_1'$
3. $s\&s_1 \models_\beta \Theta$ *and* $h\&h_1 \models_\beta \mathcal{I}$.
4. *There exists* $\theta_0' \in \Theta'$ *such that* $s' \models ant(\theta_0')$ *and* $s_1' \models ant(\theta_0')$.

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 $\mathbf{new}$ 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 $true \Rightarrow 0\ltimes$.

Theorem 4.2 is proved in Appendix B, 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 the role of the R component and reads as follows:

Lemma 4.3 (Totality and Write Confinement)
Assume $[VC]\{\Theta\}\ (R) \Longleftarrow S\ \{\Theta'\}$ *. Then* $dom(R) = \Theta$ *and* $ran(R) = \Theta'$*. Given* $\theta' \in \Theta'$*, there exists at most one* θ *such that* $(\theta, u, \theta') \in R$*. If there exists such* θ*, then* $con(\theta) = con(\theta')$*, and with* $E = con(\theta)$ *we have*

- *if* $s,\_\ [\![S]\!]\ s',\_$ *then* s *agrees with* s' *on* $\mathrm{fv}(E)$*;*
- *if* $s,r\ [\![S]\!]\ s',r'$ *(thus* S *is of form* RS*) then also* r *agrees with* r' *on* $\mathrm{ff}(E)$*.*

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 $\phi \Rightarrow E\ltimes$ 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 $\mathbf{skip}$ 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 $\mathbf{assert}$ 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 $\mathbf{if}\ B\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2$, assume that $\phi \Rightarrow E\ltimes$ occurs in Θ'. If $(\phi \Rightarrow E\ltimes) \in \Theta_u'$, we can assume from Lemma 4.3 that neither S_1 nor S_2 has modified E, and that the precondition of each S_i will contain a 2-assertion of the form $\phi_i \Rightarrow E\ltimes$; these can now be combined by R_0 into one single precondition. On the other hand, if $(\phi \Rightarrow E\ltimes) \in \Theta'_m$ 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 $\{x\ltimes,\ odd(i) \Rightarrow v\ltimes\}$. (We write $x\ltimes$ for $true \Rightarrow x\ltimes$.) Working backwards, the assignment to i transforms $odd(i) \Rightarrow v\ltimes$ to $odd(i+1) \Rightarrow v\ltimes$, which amounts to $\neg odd(i) \Rightarrow v\ltimes$, but keeps $x\ltimes$ unchanged. To process the conditional, we apply VCgen to the branches; the **else** branch produces R_2 given by

$$(x\ltimes, u, x\ltimes),$$
$$(\neg odd(i) \Rightarrow x\ltimes, m, \neg odd(i) \Rightarrow v\ltimes)$$

while the **then** branch produces R_1 given by

$$(x\ltimes, u, x\ltimes),$$
$$(\neg odd(i) \Rightarrow (v+h)\ltimes, m, \neg odd(i) \Rightarrow v\ltimes)$$

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

$$(x\ltimes, u, x\ltimes),$$
$$(\neg odd(i) \Rightarrow x\ltimes, m, odd(i) \Rightarrow v\ltimes)$$

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

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 4.5. VC_3 demands that Θ_m contains an assertion θ_m with a "weakest" antecedent. (This is no serious restriction, since if $\Theta_m = \{\phi_i \Rightarrow E_i\ltimes \mid i \in \{1 \ldots n\}\}$ we can just add $(\phi_1 \vee \ldots \vee \phi_n) \Rightarrow 0\ltimes$ to Θ_m.)

Lemma 4.5 *Assume* $[VC]\{\Theta\}\ (R) \Longleftarrow S\,\{\Theta'\}$ *with* $\models VC$. *Given* $\theta' \in \Theta'$, *there exists* $(\theta, \_, \theta') \in R$ *such that*

- *if* $S = RS$: *whenever* $s,r\ [\![S]\!]\ s',r'$ *and* $s',r' \models ant(\theta')$ *then* $s,r \models ant(\theta)$;

- *if $S = TS$: whenever s,h $[\![S]\!]$ s',h' and $s' \models ant(\theta')$ then $s \models ant(\theta)$.*

*For $S =$ **while** B **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 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 $\mathit{ff}^+ : \mathbf{1Assert} \to \mathbf{1Assert}$ such that if $\phi' = \mathit{ff}^+(\phi)$ then *(i)* ϕ' is field-free, and *(ii)* ϕ logically implies ϕ'. These demands are trivially fulfilled if $\mathit{ff}^+(\phi) = \mathit{true}$ 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) contain $x = 7 \Rightarrow y\ltimes$ if Θ_0 contains $(x = 7 \wedge \neg(.f = 8)) \Rightarrow y\ltimes$.

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 $\mathit{true} \Rightarrow (.f + x)\ltimes$ might need to be replaced by the two assertions $\mathit{true} \Rightarrow x\ltimes$ and $\mathit{true} \Rightarrow .f\ltimes$ 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 4.6 *Assume $[VC]\{\Theta\}\ (R) \Longleftarrow TS\,\{\Theta'\}$ with $\models VC$, and that $\theta' \in \Theta'$ is such that if $(\_, \gamma, \theta') \in R$ then $\gamma = u$. For $(\phi_0 \Rightarrow E_0\ltimes) \in \mathcal{I}$, if s,h $[\![TS]\!]$ s',h' then for all $l \in dom(h)$:*

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

To see why Lemma 4.6 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 4.6 now further ensures that two such diverting runs do not invalidate object flow invariants.

The clause for **new** 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 $.f = 1 \Rightarrow y\ltimes$, we generate the (trivial) precondition $0 = 1 \Rightarrow y\ltimes$; if Θ_0 contains say $\mathit{true} \Rightarrow (.f + y)\ltimes$, we generate the precondition $\mathit{true} \Rightarrow (0 + y)\ltimes$. 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, it can never hold that $x = A$, unless $A = x$. This is formalized by the function $rm_x : \mathbf{1Assert} \to \mathbf{1Assert}$ 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 $x \neq A$ and hence $x \notin \mathrm{fv}(A)$, and which maps any B not containing x to itself. Concerning the consequents, we exploit that two runs will always agree on the value of x after allocation (as β can be extended to relate the fresh locations); this is formalized by the function $rm_x : \mathbf{Exp} \to \mathbf{Exp}$ which maps E into 0 if $x \in \mathrm{fv}(E)$, and into E otherwise.

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. Example 4.4) to recognize when a 2-assertion has an antecedent which is always false, or when it is of the form $\phi \Rightarrow B\ltimes$ 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 a bit more formal about what must hold, apart from $\{\theta_1, \ldots, \theta_n\} \rhd^2 \theta$, when θ is replaced by $\theta_1 \ldots \theta_n$. Lemma 4.5 requires that for at least one $i \in \{1 \ldots n\}$ we can verify $ant(\theta) \rhd^1 ant(\theta_i)$. Moreover, we need to record in R that θ is related to each θ_i, and if we want to assign the tag u we must demand (due to Lemma 4.3) that $n = 1$ and $con(\theta) = con(\theta_1)$. These considerations suggest that rather than eliminating a 2-assertion which is always true, we replace it by a designated such assertion, e.g., $true \Rightarrow 0\ltimes$.

5 Worked Out Example

In this section we work out the examples given in Sec. 2, starting with Fig. 1(b). We want to prove that the program satisfies the specification $\{true \Rightarrow x\ltimes\}$ _ $\{odd(i) \Rightarrow result\ltimes\}$. The object invariant, $\mathcal{I}$, is a conjunction of $odd(.idx) \Rightarrow .val\ltimes$ and $odd(.idx) \Rightarrow .src\ltimes$.

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 $odd(i) \Rightarrow result\ltimes$ conjoined with the object invariant. Using VCgen's clauses for assignment, field update, and **assert**, this yields an empty set of VCs, and R_0 containing

$$\begin{array}{l}(odd(i) \wedge (.idx = i) \Rightarrow q\ltimes, m, odd(i) \Rightarrow result\ltimes) \\ (odd(.idx) \wedge (.idx = i) \Rightarrow q\ltimes, m, odd(.idx) \Rightarrow .val\ltimes) \\ (odd(.idx) \wedge (.idx = i) \Rightarrow .src\ltimes, u, odd(.idx) \Rightarrow .src\ltimes)\end{array}$$

Applying the clause in VCgen for **open** now generates the verification conditions: $VC_1 = \{odd(.idx) \wedge (.idx = i) \rhd^1 odd(.idx)\}$ and $VC_2 = \{\}$. (To see why VC_2 is empty, note that the relevant assertions are of the form $\_ \Rightarrow q\ltimes$ but $q\ltimes$ 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).

$$\begin{array}{rcl}R_1 &=& \{(odd(i) \Rightarrow x\ltimes, m, odd(i) \Rightarrow result\ltimes)\} \\ R_2 &=& \{true \Rightarrow x\ltimes, m, odd(i) \Rightarrow result\ltimes)\} \\ R_3 &=& \{(odd(i) \Rightarrow q\ltimes, m, odd(i) \Rightarrow result\ltimes)\}\end{array}$$

We have assumed that ff^+ maps $odd(.idx) \wedge (.idx = i)$ into $odd(i)$. Now the precondition of lines 9–13 can be read off from the above sets as

$$\{odd(i) \Rightarrow x\ltimes, x\ltimes, odd(i) \Rightarrow q\ltimes\}$$

where the first assertion can be removed as it follows from the second.

(**TA: The below calculations need to be checked**)

Next, we analyze lines 5–8 of Fig. 1(b) with the above as postcondition.

For lines 6–7, apart from the precondition, VCgen also generates the , and the following R_0 set which is the union of R':

$$\begin{array}{l}\{(odd(i) \wedge (odd(i) \to odd(.idx)) \Rightarrow x\ltimes,\ u,\ odd(i) \Rightarrow x\ltimes),\\ ((odd(i) \to odd(.idx)) \Rightarrow x\ltimes,\ u,\ true \Rightarrow x\ltimes)\\ (odd(i) \wedge (odd(i) \to odd(.idx)) \Rightarrow .val\ltimes,\ m,\ odd(i) \Rightarrow q\ltimes)\}\end{array}$$

and $R_{\mathcal{I}}$:

$$\begin{array}{l}\{(odd(.idx) \wedge (odd(i) \to odd(.idx)) \Rightarrow .val\ltimes,\ u,\ odd(.idx) \Rightarrow .val\ltimes),\\ (odd(.idx) \wedge (odd(i) \to odd(.idx)) \Rightarrow .src\ltimes,\ u,\ odd(.idx) \Rightarrow .src\ltimes)\}\end{array}$$

Now, using the case of **open** . . . **close**, VCgen generates the verification conditions: $VC_1 = \{odd(.idx) \wedge (odd(i) \to od$ $odd(.idx)\}$ and $VC_2 = \{\}$. Thus $VC = VC_1$.

The set R that will be used in the generation of the precondition, ???, is the union of the sets $R_1, \ldots, R_4$ below.

$$\begin{array}{lcl}R_1 & = & \{(odd(i) \Rightarrow y\ltimes, m, odd(i) \Rightarrow q\ltimes)\}\\ R_2 & = & \{\}\\ R_3 & = & \{\}\\ R_4 & = & \{(odd(i) \Rightarrow x\ltimes,\ u,\ odd(i) \Rightarrow x\ltimes), (odd(i) \Rightarrow x\ltimes,\ u,\ true \Rightarrow x\ltimes)\}\end{array}$$

Now the precondition can be read off from the above sets as:

$$\{odd(i) \Rightarrow y\ltimes, odd(i) \Rightarrow x\ltimes\}$$

Finally, we analyze lines 1–4 of Fig. 1(b) with the above as postcondition.

For lines 2–3, apart from the precondition, VCgen also generates an empty set of VCs, and the following R_0 set which is the union of R':

$$\{(odd(.idx) \Rightarrow .src\ltimes,\ m,\ odd(i) \Rightarrow y\ltimes),\ (odd(.idx) \Rightarrow x\ltimes,\ u,\ odd(i) \Rightarrow x\ltimes)\}$$

and $R_{\mathcal{I}}$:

$$\begin{array}{l}\{(odd(.idx) \Rightarrow .src\ltimes,\ u,\ odd(.idx) \Rightarrow .src\ltimes),\\ (odd(.idx) \Rightarrow .val\ltimes,\ u,\ odd(.idx) \Rightarrow .val\ltimes)\}\end{array}$$

Now, using the case of **open** . . . **close**, VCgen generates the verification conditions: $VC_1 = \{odd(.idx) \rhd^1 odd(.idx)\}$ and $VC_2 = \{\mathcal{I} \rhd^2 odd(.idx) \Rightarrow .src\ltimes\}$. Thus $VC = VC_1 \cup VC_2$.

The set R that will be used in the generation of the precondition, is the union of the sets $R_1, \ldots, R_4$ below.

$$\begin{array}{lcl}R_1 & = & \{(true \Rightarrow x\ltimes,\ m,\ odd(i) \Rightarrow y\ltimes)\}\\ R_2 & = & \{\}\\ R_3 & = & \{\}\\ R_4 & = & \{(true \Rightarrow x\ltimes,\ notm,\ odd(i) \Rightarrow x\ltimes)\}\end{array}$$

Now the overall precondition can be read off from the above sets as $true \Rightarrow x\ltimes$. We collect the VCs generated in each analysis, noting that the VCs are valid.

Fig. 7 in Appendix A shows the assertions that hold at each line in the program.

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 [23] and Naumann [21]. 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 [23] use self-composition to verify information flow automatically using the BLAST [19] tool. 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 analyses, such as a security typing. 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.

Since [23] does not address heap-manipulating programs, the work most closely related to ours is the one by Naumann [21] whose goal was the verification of information flow using existing verifiers like Spec# [7] or ESC/Java2 [13], and whose 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. [15] 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]. However, they do not provide details on how the heap is handled. Darvas et al. [14] 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 which variables are dependent on which variables; this technique forms the basis for the Spark Ada Examiner [12] which requires that each method is annotated with `derives` annotations like

```
derives u from y,z,derives w from x,y
```

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

$$[VC]\{x\ltimes, y\ltimes, z\ltimes\}\ (R) \Longleftarrow S\ \{u\ltimes, w\ltimes\}$$

with R containing the elements $(y\ltimes, \_, u\ltimes)$, $(z\ltimes, \_, u\ltimes)$, $(x\ltimes, \_, w\ltimes)$, $(y\ltimes, \_, w\ltimes)$. Our approach is more general in that it also captures *conditional* channels; we plan to investigate how to extend the Spark Ada Examiner framework to express R elements like $(i > 5 \Rightarrow y\ltimes, \_, j > 7 \Rightarrow u\ltimes)$. Also, we hope to investigate the relationship to the path conditions presented by Hammer et al. [18].

In the near future, we plan to experiment with the prototype implementation which is currently being developed by our undergraduate student Jonathan Hoag. Over the summer, we might try to integrate it with the Bogor tool [16] 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.

An important long-term goal is to develop techniques for the automatic computation of flow (loop/object) invariants, thereby moving closer to an automatic information flow analysis, and to extend the framework to an interprocedural setting. We would also like a (sound and preferably complete) axiomatization of $\rhd^2$ so as to automatically check whether the VCs generated are satisfiable; a trivial rule is that $\phi \Rightarrow x\ltimes \wedge \phi \Rightarrow w\ltimes \;\rhd^2\; \phi \Rightarrow (x+w)\ltimes$ holds for all ϕ,x,w. Relatedly, we would like to investigate whether our analysis is in some sense "optimal", with the preconditions being "weakest".

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 for 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 *PLAS*, 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 *Proceedings of CASSIS*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69, 2004.

[8] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure information flow by self-composition. In *IEEE Computer Security Foundations Workshop (CSFW)*, 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*. ACM, November 2004.

[13] David R. Cok and Joseph Kiniry. ESC/Java2: Uniting ESC/Java and JML. In *Proceedings of CASSIS*, volume 3362 of *Lecture Notes in Computer Science*, pages 108–128, 2004.

[14] Adam Darvas, Reiner Hähnle, and David Sands. A theorem proving approach to analysis of secure information flow. In *SPC*, volume 3362 of *Lecture Notes in Computer Science*, pages 151–171, 2005.

[15] Guillaume Dufay, Amy Felty, and Stan Matwin. Privacy-sensitive information flow with JML. In *CADE*, 2005.

[16] 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)*, 2005.

[17] Joseph Goguen and Jose Meseguer. Security policies and security models. In *Proc. IEEE Symp. on Security and Privacy*, pages 11–20, 1982.

[18] 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.

[19] 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.

[20] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *POPL*, 1999.

[21] David A. Naumann. From coupling relations to mated invariants for secure information flow and data abstraction. In *ESORICS*, 2006.

[22] Andrei Sabelfeld and Andrew C. Myers. A model for delimited information release. In *ISSS*, 2004.

[23] 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.

[24] 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 Example Derivation for Fig. 1(b)

B Proof of Correctness

```
 *** THIS SECTION IS CURRENTLY A ROUGH DRAFT ONLY, WITH
      MANY PARTS NOT IN LATEX; BUT THE PROOFS ARE QUITE DETAILED
     AND HAVE BEEN CHECKED AT LEAST ONCE.
```

To establish Theorem 4.2, we shall need to establish a sequence of auxiliary results, including Lemmas 4.3, 4.5, and 4.6.

B.1 Basic Results about Substitution

```
Lemma: for all A0, for all A, for all s, for all r, for all x:
   with v = [[A0]]s,r, and
  with s' = s{x->v}, we have
     [[A{x->A0}]]s,r = [[A]]s',r
 Proof: induction in A.
   * A = c:  then both sides evaluate to c
   * A = x:  then both sides evaluate to v
   * A = y, y != x: then both sides evaluate to s(y).
   * A = f: then both sides evaluate to r(f)
   * A = A1 aop A2: the induction hypothesis easily yields the claim.

Lemma: for all A0, for all A, for all s, for all r, for all f:
   with v = [[A0]]s,r, and
  with r' = r{f->v}, we have
     [[A{f->A0}]]s,r = [[A]]s,r'
Proof: similar to previous Lemma.

Lemma: for all B, for all A, for all s, for all r, for all x:
  with v = [[A]]s,r  and
  with s' = s{x->v}, we have
     [[B{x->A}]]s,r = [[B]]s',r
 Proof: First let us assume that B is of the form (A1 < A2). Then
     [[B{x->A}]]s,r = true
   iff
     [[A1{x->A}]]s,r < [[A2{x->A}]]s,r
   iff (by previous Lemma)
     [[A1]]s',r < [[A2]]s',r
   iff
     [[B]]s',r = true.

Lemma: for all B, for all A, for all s, for all r, for all f:
  with v = [[A]]s,r  and
  with r' = r{f->v}, we have
```

```
    [[B{f->A}]]s,r = [[B]]s,r'
Proof: similar to previous Lemma.

Lemma (ExpSub1)
for all E, for all A, for all s, for all r, for all x:
  with v = [[A]]s,r  and
  with s' = s{x->v}, we have
     [[E{x->A}]]s,r = [[E]]s',r
Proof: follows from previous Lemmas.

Lemma (ExpSub2)
for all E, for all A, for all s, for all r, for all f:
  with v = [[A]]s,r  and
  with r' = r{f->v}, we have
     [[E{f->A}]]s,r = [[E]]s,r'
Proof: follows from previous Lemmas.

Lemma PhiSub1: for all A, for all s, for all r, for all x:
 with v = [[A]]s,r and s' = s{x->v}, we have
   s',r |= \phi iff s,r |= \phi{x -> A}
Proof: induction in \phi.
 The base case is where \phi = B for some boolean expression B.
   Then the claim follows directly from the previous Lemma.
 The inductive cases are straightforward.

Lemma PhiSub2: for all A, for all s, for all r, for all f:
 with v = [[A]]s,r and r' = r{f->v}, we have
   s,r' |= \phi iff s,r |= \phi{f -> A}
Proof: similar to the previous Lemma
```

Results about rm_x.

```
RemL_x: 1assert -> 1assert is given by

  RemL_x(x = x) = true
  RemL_x(x = A) = false  if x notin fv(A)
  RemL_x(A = x) = false  if x notin fv(A)
  RemL_x(B) = B  if x notin B
    ## as we do not have pointer arithmetic,
    ## the base clauses are exhaustive.
  RemL_x(\phi_1 or \phi_2) = RemL_x(\phi_1) or RemL_x(\phi_2)
  RemL_x(\phi_1 and \phi_2) = RemL_x(\phi_1) and RemL_x(\phi_2)
  RemL_x(~\phi) = ~RemL_x(\phi)

Lemma: Let s' = s{x -> l}, with l \notin ran(s).
    s' |= \phi iff s |= RemL_x(\phi).
```

```
Proof:
  Induction in \phi.
  The inductive cases are straightforward. For the base cases:
 * \phi: x = x
     the claim is that s' |= x = x iff s |= true; this is obvious.
 * \phi: x = A  with x notin fv(A).
     the claim is that s' |= A = x iff s |= false,
      which from x \notin fv(A) is equivalent to
         [[A]]s \neq l
     which clearly follows from l \notin ran(s).
 * \phi: A = x (x \neq A)
     similar to the above case.
 * \phi: B, with x \notin B.
     the claim is that s' |= B iff s |= B, which is obvious.
```

Results about rm_x.

```
RemR_x: Exp -> Exp is given by

  RemR_x(E) = 0     if x occurs in E
  RemR_x(E) = E     if x does not occur in E

Lemma: Let s' = s{x -> l}, with l \notin ran(s),
   and let s'1 = s1{x -> l1}, with l1 \notin ran(s1).
 Assume that \beta' = \beta U {l,l1}, and that x occurs in E.
  Then [[E]]s' \beta' [[E]]s'1.
Proof:
   Due to the absence of pointer arithmetic,
      there are only 4 possibilities:
   * E = x
      Here [[E]]s' = l, [[E]]s'1 = l1. Thus [[E]]s' \beta' [[E]]s'1.
   * E = (x = x)
      Here [[E]]s' = true, [[E]]s'1 = true,
        and the claim is obvious as true \beta true.
   * E = (x = A) with x notin fv(A)
      Due to the absence of pointer arithmetic, and the fact that
       l notin ran(s), we infer that [[A]]s \neq l; similarly,
      we infer that [[A]]s1 \neq l1. Thus also [[A]]s' \neq l and
        [[A]]s'1 \neq l1, so since [[x]]s' = l and [[x]]s'1 = l1,
      we infer that [[E]]s' = false and [[E]]s'1 = false.
       This yields the claim, as false \beta false.
   * E = (A = x) with x notin fv(A)
       similar to the previous case.
```

B.2 Totality and Write Confinement

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

Totality $dom(R) = \Theta$ and $ran(R) = \Theta'$,

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 $s,\_\ [\![S]\!]\ s',\_$ then s agrees with s' on $\mathrm{fv}(E)$;
- if $s,r\ [\![S]\!]\ s',r'$ (thus S is of form RS) then also r agrees with r' on $\mathrm{ff}(E)$.

Proof:

```
 The proof is by induction in S
    (using the terminology from the algorithm),
  with a case analysis on S:

S = skip: trivial.

S = assert(\phi_0): trivial.

S = x := A.
 If (\theta,u,\theta') \in R with E' = \rhs(\theta') and E = \rhs(\theta)
  then x \notin \fv(E') so E = E' and the claim is obvious.

S = .f := A.
 Similar to the above case

S = S1 ; S2.
 Totality, Wellformedness, and Uniqueness are all
   obvious from the induction hypothesis.
 Now assume that (\theta,u,\theta') \in R; this happens because
   (\theta,u,\theta'') \in R_1; (\theta'',u,\theta') \in R_2.
 Inductively, we can assume that S_1 and S_2 obeys Write Confinement.
 Therefore, with E = \rhs(\theta'), we infer
        \rhs(\theta'') = E and next \rhs(\theta) = E.
  Finally, assume s,r [[S]] s',r' (the case s,h [[S]] s',h' is similar).
   Then there exists s'',r'' such that
    s,r [[S1]] s'',r'', s'',r'' [[S2]] s',r'.
  Given x in fv(E), we must show that s'(x) = s(x).
   But this follows since s'(x) = s''(x) and s(x) = s''(x).
  Similarly, we can show that if f in ff(E) then r'(f) = r(f).

S = if B then S1 else S2.
```

```
 Wellformedness follows clearly from the induction hypothesis.
 We now address Totality.
 By construction, for each \theta \in \Theta there exists
  \theta' with (\theta,_,\theta') \in R.
 Now let \theta' \in \Theta' be given. If \theta' \in \Theta'_m,
   the claim follows from R1 being total.
  Otherwise, \theta' \in \Theta'_u, and inductively
   we infer that there exists \theta_1,\theta_2 such that
     (\theta_1,u,\theta') \in R_1, (\theta_2,u,\theta') \in R_2,
   and such that \rhs(\theta_1) = \rhs(\theta_2) = \rhs(\theta').
   But this shows that there exists \theta with (\theta,_,\theta') \in R.

 Next consider \theta' with (\_,u,\theta') \in \Theta'.
  We infer that \theta' \in \Theta'_u.
  Inductively, there exists exactly one \theta_1 such that
    (\theta_1,u,\theta') \in R_1, and
       exactly one \theta_2 such that
     (\theta_2,u,\theta') \in R_2;
    moreover, with E = \rhs(\theta') we have
      rhs(\theta_1) = rhs(\theta_2) = E.
   But then we see from the algorithm (R_0) that there exists exactly
     one \theta such that (\theta,u,\theta') \in R, and \rhs(\theta) = E.
  We are left with showing that if s,r [[S]] s',r' then
      s' and s agree on fv(E), and r' and r agree on fv(E).
  Wlog, we can assume that s,r |= B,  s,r [[S1]] s',r'.
  The claim then follows from the induction hypothesis on S1.

S = new x in RS close
 Wellformedness follows by construction.
 Totality and Uniqueness follow easily from the induction hypothesis.
 Now assume that (\theta,u,\phi => E'#) \in R.
 We infer that there exists (\phi => E#, u, \phi' => E'#) \in R_0
  such that \theta = Rem_x(\phi{f->default}) => Rem_x(E{f->default})
      and that x \notin fv(E).
  Inductively we infer that E' = E;
   since E' is field-free, we infer that E = E{f->default(f)};
  since x \notin fv(E), we infer the desired \rhs(\theta) = E = E'.
 Finally, we must show that if s,h [[S]] s',h' then s and s' agree on fv(E).
  So assume that with r = default we have s{x->l},r [[RS]] s',r'.
   Inductively, we infer that s{x->l},s' agree on fv(E).
  As x \notin fv(E), this amounts to the desired result:
   that s,s' agree on fv(E).

S = open x in RS close
  Concerning Totality, R is total on \Theta by construction;
```

```
   that R is total on \Theta' follows by the induction hypothesis,
    using R1 and R4.
  Concerning Wellformedness, the only issue is R4,
   but since we can assume inductively that RS satisfies WriteConfinement,
    we infer that E = \rhs(\theta') and hence E is field-free if \Theta' is.
  Concerning Uniqueness, the only relevant clause is R4,
   and the claim follows since inductively, RS satisfies uniqueness.
  Now assume that  (\theta,u,\theta') \in R with \rhs(\theta') = E.
   From R4 we see that there exists (\theta_0,u,\theta') \in R_0 with
    rhs(\theta_0) = rhs(\theta). Inductively, we infer that
      rhs(\theta_0) = E, and hence \rhs(\theta) = E, as desired.
  Finally, assume that s,h [[S]] s',h'.
    because h' = h{l->r'} where with r = h(l) we have
    s,r [[RS]] s',r'.
  Inductively, s and s' agree on fv(E), as desired.

S = while B do S0.
 We shall only consider the case where S is a top-level command;
   the other case is similar.
 Totality, Wellformedness, and Uniqueness are trivial.
 Now assume that (\theta,u,\theta') in R,
   we infer \theta = \theta'. Let E = rhs(\theta).
 We have \theta \in \Theta_u, so there exists no (_,m,\theta) \in R_0.
 Inductively, R_0 is total, so there exists (_,u,\theta) \in R_0.
 Inductively on R_0, we thus infer that if s,h [[S0]] s',h' then
  s,s' agree on fv(E). It is now easy to show by induction in i
  that if s,h f_i s',h' then s,s' agree on fv(E).
```

□

B.3 Other Key Lemmas

Lemma 4.5 Assume $[VC]\{\Theta\}\ (R) \Longleftarrow S\ \{\Theta'\}$ with $\models VC$. Given $\theta' \in \Theta'$, there exists $(\theta, \_, \theta') \in R$ such that

- if $S = RS$: whenever $s,r\ [\![S]\!]\ s',r'$ and $s',r' \models ant(\theta')$ then $s,r \models ant(\theta)$;
- if $S = TS$: whenever $s,h\ [\![S]\!]\ s',h'$ and $s' \models ant(\theta')$ then $s \models ant(\theta)$.

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

Proof:

```
We prove this by induction in S
   (using the terminology from the algorithm),
  with a case analysis on S:

We first consider the case where S = RS.
```

```
We define Q(RS,\theta,\theta’) as the following property:
   whenever s,r [[RS]] s’,r’, and
    s’,r’ |= \lhs(\theta’), then s,r |= \lhs(\theta).
 The claim is now that given \theta’ \in \Theta’,
   there exists (\theta,_,\theta’) \in R with Q(RS,\theta,\theta’).

RS = skip: trivial.

RS = assert(\phi_0).
 Given \theta’, there exists \theta with (\theta,_,\theta’) \in R
    such that \lhs(\theta) = \lhs(\theta’) /\ \phi_0.
 We shall now show Q(RS,\theta,\theta’):
 if s,r [[RS]] s’,r’  then s,r |= \phi_0, and s’ = s and r’ = r.
 But then s’,r’ |= \lhs(\theta’) clearly implies
    s,r |= \lhs(\theta), as desired.

RS = x := A.
  Here r’ = r; s’ = s{x -> v} where v = [[A]]s,r.
  The claim is that s’,r’ |= \phi implies s,r |= \phi{x -> A}.
  But this follows from Lemma PhiSub1.

RS = .f := A.
  Here s’ = s; r’ = r{x -> v} where v = [[A]]s,r.
  The claim is that s’,r’ |= \phi implies s,r |= \phi{.f -> A}.
  But this follows from Lemma PhiSub2.

RS = RS1 ; RS2.
 Given \theta’ \in \Theta’. Inductively on RS2 there exists
  \theta’’ with (\theta’’,_,\theta’) \in R_2 and with Q(RS2,\theta’’,\theta’).
 Then, inductively on RS1, there exists
  \theta with (\theta,_,\theta’’) \in R_1 and with Q(RS1,\theta,\theta’’).
   Note that (\theta,_\,\theta’) \in R.
 We shall now show Q(RS,\theta,\theta’).
  So assume that s,r [[RS]] s’,r’, that is,
   there exists s’’,r’’ such that
    s,r [[RS1]] s’’,r’’, s’’,r’’ [[RS2]] s’,r’.
 Further assume that s’,r’ |= \lhs(\theta’). From Q(RS2,\theta’’,\theta’)
   we infer that s’’,r’’ |= \lhs(\theta’’).
  From Q(RS1,\theta,\theta’’) we next infer the desired
     s,r |= \lhs(\theta).

RS = if B then RS1 else RS2.
 Given \theta’ \in \Theta’, with \theta’ = \phi’ => E’#.
   Inductively on RS1 and RS2, there exists
   \theta_1 with (\theta_1,_,\theta’) \in R_1 and Q(RS1,\theta_1,\theta’), and
```

```
   \theta_2 with (\theta_2,_,\theta') \in R_2 and Q(RS2,\theta_1,\theta').
   Let \theta_1 = \phi_1 => E_1#, and \theta_2 = \phi_2 => E_2#.
   Define \phi = \phi_1 /\ B \/ \phi_2 /\ ~B.
   We now define \theta:
    * if \theta' \in \Theta'_m we define \theta = \phi => B#;
    * if \theta' \in \Theta'_u, in which case Lemma Write Confinement
        says that E_1 = E_2 = E,
       we define \theta = \phi => E#.
   We clearly have (\theta,_,\theta') \in R,
     and must prove Q(RS,\theta,\theta').
   So assume that s,r [[RS]] s',r', and that s',r' |= \phi'.
  Wlog, we can assume that s,r |= B,  s,r [[RS1]] s',r'.
  From Q(RS1,\theta_1,\theta') we infer that s,r |= \phi_1.
  But then s,r |= \phi_1 /\ B and hence s,r |= \phi, as desired.

RS = while B do RS0
 As the similar case for top-level commands

We next consider the case where S = TS.
We define Q(TS,\theta,\theta') as the following property:
   whenever s,h [[RS]] s',h', and
    s' |= \lhs(\theta'), then s |= \lhs(\theta).
 The claim is now that given \theta' \in \Theta',
   there exists (\theta,_,\theta') \in R with Q(TS,\theta,\theta')
 (and that if TS = while ... then \theta is given explicitly in a certain way).

TS = skip: trivial.

TS = assert(\phi_0).
  as the similar case for RS

S = x := A.
  as the similar case for RS

S = S1 ; S2.
  as the similar case for RS

S = if B then S1 else S2.
  as the similar case for RS

S = while B do S0
 First consider the case when \theta' \in \Theta_u.
Then (\theta',_,\theta') \in R,
so with \theta' = \phi' => _ is sufficient to prove that if s,h f_i s',h'
then s' |= \phi' implies s |= \phi'.
```

We shall do so by an inner induction in i. For i = 0, we have s = s’
and the claim is obvious. Otherwise, we have s,h [[S0]] s’’,h’’ and
s’’,h’’ f_{i-1} s’,h’. Now assume s’ |= \phi’. By the inner induction,
we have s’’ |= \phi’. Note that there is no (_,m,\theta’) \in R_0,
so by Lemma (Write Confinement & Totality) we infer that there exists exactly
one \theta such that (\theta_,\theta’) \in R_0.
Inductively on S_0,
we now infer that with \theta = \phi => E# we have s |= \phi.
But \logimpone{\phi}{\phi’} \in VC_5 \subseteq VC,
so from |= VC we infer that s |= \phi’, as desired.

Next consider the case when \theta’ \in \Theta_m.
Then (\theta_m,_,\theta’) \in R_m.
Let \theta’ = \phi’ => _ and \theta_m = \phi_m => _.
Since \logimpone{\phi’}{\phi_m} \in VC_3 \subseteq VC,
we from |= VC infer that s’ |= \phi’ implies s’ |= \phi_m.
It is thus sufficient to prove that if s,h f_i s’,h’
then s’ |= \phi_m implies s |= \phi_m, and shall do
so by an inner induction in i. For i = 0, we have s = s’ and
the claim is obvious. Otherwise, we have s,h [[S0]] s’’,h’’ and
s’’,h’’ f_{i-1} s’,h’. Now assume s’ |= \phi_m. By the inner induction,
we have s’’ |= \phi_m. Inductively on S0, there exists
(\phi => _,_,\theta_m) \in R_0 such that s |= \phi.
Since \logimpone{\phi}{\phi_m} \in VC_4 \subseteq VC,
we from |= VC infer s |= \phi_m, as desired.

S = new x in RS close
Given \theta’ \in \Theta’, with \theta’ = \phi’ => _.
Inductively on RS, there exists (\theta_0, _, \theta’) \in R0
with Q(RS,\theta_0,\theta’). Let \theta_0 = \phi_0 => _.
With \theta = \phi => _ where \phi = RemL_x(\phi_0[f->default]),
we have (\theta, _, \theta’) \in R;
we shall prove that Q(S,\theta,\theta’).
So assume that s,h [[S]] s’,h’ because
with r = default there exists l with l \notin ran(s) such that
s{x->l},r [[RS]] s’,r’.
where h’ = h{l->r’}.
Assume that s’ |= \phi’.
From Q(RS,\theta_0,\theta’) we have s{x->l},r |= \phi_0.
By (repeated application of) Lemma PhiSub2 we infer that
Then clearly s{x->l} |= \phi_0[f -> default]
By the Lemma about RemL_x, this implies the desired s |= \phi.

S = open x in RS close
Given \theta’ \in \Theta’, with \theta’ = \phi’ => E’#.

```
  By the induction hypothesis, there exists
    (\phi_0 => _,_,\theta') \in R_0 such that
    whenever s,r [[RS]] s',r' and s',r' |= \phi' then s,r |= \phi_0.
    By either R1 or R4, we infer that there exists \phi with
      \phi = RemF+(\phi_0) such that
         (\phi => _, _,\theta') \in R.
   Now assume that s,h [[S]] s',h' with s' |= \phi'.
       Then s,h(l) [[RS]] s',h'(l), so we infer that s,h(l) |= \phi_0
      and thus s |= \phi, as desired.
```

□

Lemma 4.6 Assume $[VC]\{\Theta\}\ (R) \Longleftarrow TS\,\{\Theta'\}$ with $\models VC$, and that $\theta' \in \Theta'$ is such that if $(\_, \gamma, \theta') \in R$ then $\gamma = u$. For $(\phi_0 \Rightarrow E_0 \ltimes) \in \mathcal{I}$, if s,h $[\![TS]\!]$ s',h' then for all $l \in dom(h)$

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

Proof:

```
We assume that \phi_0 => E_0# \in I has been given,
and define Q(R) to mean:
  if s,h R s',h' then for all l in dom(h), for all f in ff(E_0):
    * if h'(l) |= \phi_0 then h(l) |= \phi_0,
    * h(l)(f) = h'(l)(f).
The claim is now that if
$\algo{TS}{\Tht'}{\Tht}{R}{VC}$ with $\satvc{VC}$,
and \theta' \in \Theta' is such that no (_,m,\theta') \in R,
then Q(TS).

We shall prove that by induction in TS,
using the terminology from the algorithm,
and do a case analysis on TS.

The following cases are all trivial, as h' = h:
 TS = skip, TS = assert(\phi_0), TS = x := A:

TS = TS1 ; TS2.
  We have s'',h'' such that s,h [[TS1]] s'',h''; s'',h'' [[TS2]] s',h'.
  Our assumption is that \theta' \in \Theta' where no (_,m,\theta') \in R.
  By Lemma on Totality, we infer that no (_,m,\theta') \in R_2.
    and that (\theta'',u,\theta') \in R_2 for some \theta''.
   Clearly, there can be no (_,m,\theta'') \in R_1.
  Inductively, we can thus assume that Q([[TS1]]) and Q([[TS2]]).
  Given l in dom(h); note that l in dom(h'').
  * if h'(l) |= \phi_0 then we infer from Q([[TS2]]) that
    h''(l) |= \phi_0, and then from Q([[TS1]]) that h(l) |= \phi_0.
```

```
  * given f in ff(E_0), we infer from Q([[TS1]]) that
     h(l)(f) = h''(l)(f), and infer from Q([[TS2]]) that
     h''(l)(f) = h'(l)(f), yielding the desired h(l)(f) = h'(l)(f).

TS = if B then TS1 else TS2
  Wlog. we can assume that s,h [[TS]] s',h' because [[B]]s = true
     and s,h [[TS1]] s',h'.
  Our assumption is that \theta' \in \Theta' with no (_,m,\theta') \in R.
  Thus \theta' \in \Theta'_u, and therefore there is no (_,m,\theta') \in R_1.
  Hence we can apply the induction hypothesis on TS1 to give us
    Q([[TS1]]) which clearly giving us the desired claim.

TS = new x in RS close
 If l in dom(h) then h'(l) = h(l), and the claim is clear.

TS = open x in RS close
 Assume s,h [[S]] s',h' because with s(x) = l, r = h(l) we have
    s,r [[RS]] s',r',  h' = h{l -> r'}.
  Given l' \in dom(h),
    we must prove that if h'(l') |= \phi_0 then h(l') |= \phi_0,
    and that for all f in ff(E_0), h(l')(f) = h'(l')(f).
    If l' \neq l, the claim is obvious, as then h'(l') = h(l').
   So assume that l' = l; we must prove that
     *1   if r' |= \phi_0 then r |= \phi_0
     *2   for all f in ff(E_0), r(f) = r'(f).
 Our assumption is that for some \theta' \in \Theta',
   there exists no (_,m,\theta') \in R.
 From R2 we therefore infer that there
    exists no (_,m,\phi_0 => E_0#) \in R_0.
 We can now apply Write Confinement to RS and infer that
 there exists exactly one \theta_0, of the form \phi'_0 => E_0#, such that
    (\theta_0,u,\phi_0 => E_0#) \in R_0, and that
    r and r' agree on ff(E_0), yielding *2.
  We now address *1, and thus assume that r' |= \phi_0.
   From Lemma BackSatExists applied
    to RS we infer that s,r |= \phi'_0.
 Since LogImpOne{\phi'_0}{\phi_0} \in VC_1 \subseteq VC,
     we from |= VC infer s,r |= \phi_0 which
    (since \phi_0 is an object assertion) amounts to the desired r |= \phi_0.

TS = while B do TS0.
Assume that \theta' is such that there exists no (_,m,\theta') \in R.
Then \theta' in \Theta_u, so there exists  no (_,m,\theta') \in R_0.
Inductively on TS0, we infer Q([[TS0]]).
Our task is done if we can prove Q(f_i), which we shall do by
```

```
induction in i.
For i = 0, the claim is obvious as h' = h.
 Now assume that s,h f_{i+1} s',h' because s,h [[TS0]] s'',h'' and
    s'',h'' f_i s',h'. Let l \in dom(h), then (by Lemma) also l \in dom(h'').
    If h'(l) |= \phi_0 then, by induction, we have h''(l) |= \phi_0,
     and from Q([[TS0]]) even h(l) |= \phi_0.
    Given f in ff(E_0), by induction we have h''(l)(f) = h'(l)(f),
     and from Q([[TS0]]) we have h(l)(f) = h''(l)(f),
    implying the desired h(l)(f) = h'(l)(f).
```

□

B.4 Correctness of Record Commands

Proposition 4.1 Assume that

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

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

Proof:

```
Proof: induction in RS, using the terminology from the algorithm,
  with a case analysis on RS:

RS = skip: trivial.

RS = assert(\phi_0):
 We have s,r |= \phi_0, s1,r1 |= \phi_0, r' = r, s' = s, r'1 = r1, s'1 = s1.
 Let \phi => E# \in \Theta' be given,
  and assume that s,r |= \phi and s1,r1 |= \phi;
 we must prove [[E]]s,r \beta [[E]]s1,r1.
 But from s,r&s1,r1 |= \Theta  we have s,r&s1,r1 |= (\phi /\ \phi_0) => E#
  and since s,r |= \phi /\ \phi_0 and s1,r1 |= \phi /\ \phi_0 this implies
   the desired [[E]]s,r \beta [[E]]s1,r1.

RS = x := A.
 Given \theta' = \phi' => E'# \in \Theta,
  and assume that s',r' |= \phi' and s'1,r'1 |= \phi'
   so as to prove [[E']]s',r' \beta [[E']]s'1,r'1.
 Here s' = s{x -> v} with v = [[A]]s,r, r' = r;
    s'1 = s1{x -> v1} with v1 = [[A]]s1,r1, r'1 = r1.
 With \phi = \phi'{x -> A} and E = E'{x -> A} we have
   \phi => E \in \Theta and thus s,r&s1,r1 |= \phi => E#.
```

```
 From Lemma PhiSub1 we infer from s’,r’ |= \phi’ and s’1,r’1 |= \phi’
  that s,r |= \phi and s1,r1 |= \phi, so from s,r&s1,r1 |= \phi => E#
 we infer that [[E]]s,r \beta [[E]]s1,r1.
 By Lemma ExpSub1, we now infer [[E’]]s’,r’ = [[E]]s,r and
    [[E’]]s’1,r’1 = [[E]]s1,r1. Hence we get the desired
    [[E’]]s’,r’ \beta [[E’]]s’1,r’1.

RS = .f := A.
 Similar to the previous case, using Lemma PhiSub2 rather than PhiSub1,
 and Lemma ExpSub2 rather than ExpSub1.

RS = RS1 ; RS2.
 There exists s’’,r’’ and s’’1,r’’1 such that
   s,r [[RS1]] s’’,r’’ and s’’,r’’ [[RS2]] s’,r’ and
   s1,r1 [[RS1]] s’’1,r’’1 and s’’1,r’’1 [[RS2]] s’,r’.
 Given s,r&s1,r1 |=\beta \Theta, we can apply the induction hypothesis on RS1
   to give s’’,r’’&s’’1,r’’1 |=\beta \Theta’’, and next
   apply the induction hypothesis on RS2 to give the desired
      s’,r’&s’1,r’1 |=\beta \Theta’.

RS = if B then RS1 else RS2.
  Assume that s,r [[RS]] s’,r’, and s1,r1 [[RS]] s’1,r’1.
  Assume that s,r&s1,r1 |=\beta \Theta.
  We must prove s’,r’&s’1,r’1 |=\beta \Theta’.
  Except for symmetry, there are two cases:
  * [[B]]s,r = [[B]]s1,r1 = true.
      Then s,r [[RS1]] s’,r’ and s1,r1 [[RS1]] s’1,r’1.
    We shall now show s&s1 |= \Theta_1.
    So given \phi_1 => E_1# \in \Theta_1,
    and assume s,r |= \phi_1, s1,r1 |= \phi_1;
       our obligation is to show [[E_1]]s,r \beta [[E_1]]s1,r1.
    By Lemma Totality, there exists \theta’ such that
            (\phi_1 => E_1#,_,\theta’) \in R_1.
    Two cases:
      \theta’ \in \Theta’_m
         Then by R’1, (\phi_1 /\ B => E_1#) \in \Theta.
      \theta’ \in \Theta’_u
         By Write Confinement, and R_0,
        we infer that there exists \phi_2 such that
        (\phi_1 /\ B) \/ (\phi_2 /\ ~B) => E_1# \in \Theta.
    Since s,r |= \phi_1 /\ B,  s1,r1 |= \phi_1 /\ B
      and since s,r&s1,r1 |= \Theta,
     we in both cases infer the desired [[E_1]]s,r \beta [[E_1]]s1,r1.

   Having established s,r&s1,r1 |=\beta \Theta_1,
```

```
      by induction on RS1
      we have the desired s',r'&s'1,r'1 |=\beta \Theta'.

  * s,r |= ~B  s1,r1 |= B
    Then s,r [[RS2]] s',r' and  s1,r1 [[RS1]] s'1,r'1
    Given \theta' = \phi' => E# \in \Theta',
      and assuming s',r' |= \phi' and s'1,r'1 |= \phi',
      our proof obligation is to show [[E]]s',r' \beta [[E]]s'1,r'1.
    We shall establish that \theta' \in \Theta'_u,
      by showing that \theta' \in \Theta'_m leads to a contradiction:
      By Lemma BackSatExists applied to RS1 and RS2,
         we infer that there exists
            (\phi_1 => _,_,\theta') \in R_1,
            (\phi_2 => _,_,\theta') \in R_2,
        such that s1,r1 |= \phi_1, s,r |= \phi_2.
      By construction of \Theta, the clause R'_0,
             (\phi_1 /\ B) \/ (\phi_2 /\ ~B) => B# \in \Theta.
      So since s,r |= (\phi_1 /\ B) \/ (\phi_2 /\ ~B)
          and s1,r1 |= (\phi_1 /\ B) \/ (\phi_2 /\ ~B),
          and s,r&s1,r1 |=\beta \Theta, we infer [[B]]s,r \beta [[B]]s1,r1.
      But this contradicts s,r |= ~B and s1,r1 |= B.

    We have established \theta' \in \Theta'_u.
      By Write-Confinement on RS, there exists unique
            (\phi => E#,\_,\theta') \in R
       and s,s' agree on fv(E); s1,s'1 agree on fv(E)
       and r,r' agree on ff(E); r1,r'1 agree on ff(E).
     By Lemma BackSatExists, we infer that
            s,r |= \phi    s1,r1 |= \phi.
      From s,r&s1,r1 |=\beta \Theta we thus infer [[E]]s,r \beta [[E]]s1,r1
        But since [[E]]s,r = [[E]]s',r' and [[E]]s1,r1 = [[E]]s'1,r'1,
     this amounts to the desired [[E]]s',r' \beta [[E]]s'1,r'1.

RS = while B do RS0
  As the similar case for TS
```

□

B.5 Correctness of Top-level Commands

Theorem 4.2 Assume that

1. $[VC]\{\Theta\}\ (\_) \Longleftarrow TS\ \{\Theta'\}$ and that $\models VC$
2. $s,h\ [\![RS]\!]\ s',h'$ and that $s_1,h_1\ [\![RS]\!]\ s'_1,h'_1$
3. $s\&s_1 \models_\beta \Theta$ and $h\&h_1 \models_\beta \mathcal{I}$.

4. There exists $\theta'_0 \in \Theta'$ such that $s' \models ant(\theta'_0)$ and $s'_1 \models ant(\theta'_0)$.

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

Proof:

```
Proof: induction in TS, using the terminology there,
  with a case analysis on TS:

TS = skip
  Obvious, with \beta' = \beta.

TS = assert(\phi_0)
 We have s |= \phi_0, s1 |= \phi_0, h' = h, s' = s, h'1 = h1, s'1 = s1.
 We shall prove the claim with \beta' = \beta;
  the only non-trivial point is that s'&s'1 |=\beta \Theta'.
 Let \phi => E# \in \Theta' be given,
  and assume that s |= \phi and s1 |= \phi;
 we must prove [[E]]s \beta [[E]]s1.
 But from s&s1 |= \Theta and (\phi /\ \phi_0) => E# \in \Theta we infer
    s&s1 |= \phi /\ \phi_0 => E#,
 which yields the claim since s |= \phi /\ \phi_0 and s1 |= \phi /\ \phi_0.

TS = x := A
 We shall prove the claim with \beta' = \beta;
  the only non-trivial point is that s'&s'1 |=\beta \Theta',
  so consider \theta' \in \Theta'.
  Let \theta' = \phi' => E'#,
   and assume that s' |= \phi' and s'1 |= \phi'
   so as to prove [[E']]s' \beta [[E']]s'1.
 Here s' = s{x -> v} with v = [[A]]s,
    s'1 = s1{x -> v1} with v1 = [[A]]s1.
 With \phi = \phi'{x -> A} and E = E'{x -> A} we have
   \phi => E \in \Theta and thus s&s1 |= \phi => E#.
 From Lemma PhiSub1 we infer from s' |= \phi' and s'1 |= \phi'
  that s |= \phi and s1 |= \phi, so from s&s1 |= \phi => E#
 we infer that [[E]]s \beta [[E]]s1.
 By Lemma ExpSub1, we now infer [[E']]s' = [[E]]s and
    [[E']]s'1 = [[E]]s1, yielding the desired
    [[E']]s' \beta [[E']]s'1.

TS = TS1; TS2
 Given s&s1 |=\beta \Theta and h&h1 |=\beta I.
 There exists s'',h'' and s''1,h''1 such that
   s,h [[TS1]] s'',h'' and s'',h'' [[TS2]] s',h' and
   s1,h1 [[TS1]] s''1,h''1 and s''1,h''1 [[TS2]] s',h'.
 Our assumptions also are that there
```

```
   exists \theta'_0 \in \Theta' such that s' |= lhs(\theta'_0)
    and s'1 |= lhs(\theta'_0); by Lemma BackSatExists we infer
   that there exists \theta''_0 \in \Theta'' such that
    s''1 |= lhs(\theta''_0) and s'' |= lhs(\theta''_0).
 We can thus apply the induction hypothesis on TS1 to find
   \beta'' over h'',h''1 extending \beta such that
   s''&s''1 |=\beta'' \Theta'' and h''&h''1 |=\beta'' I.
 Next we can apply the induction hypothesis on TS2 to find
 \beta' over h',h'1 extending \beta'' such that
     s'&s'1 |=\beta' \Theta' and h'&h'1 |=\beta' I.
 This is as desired, since \beta' extends \beta.

TS = if B then TS1 else TS2
  Assume that s,h [[TS]] s',h' and s1,h1 [[TS]] s'1,h'1.
  Assume that s&s1 |=\beta \Theta and h&h1 |=\beta I
  Except for symmetry, there are two cases:
  * [[B]]s = [[B]]s1 = true.
      Then s,h [[TS1]] s',h' and s1,h1 [[TS1]] s'1,h'1.
    We shall now show s&s1 |= \Theta_1.
    So given \phi_1 => E_1# \in \Theta_1,
    and assume s |= \phi_1, s1 |= \phi_1;
       our obligation is to show [[E_1]]s \beta [[E_1]]s1.
     By Write Confinement (Totality), there exists \theta' such that
            (\phi_1 => E_1#,_,\theta') \in R_1.
    Two cases:
      \theta' \in \Theta'_m
         Then by R'1, (\phi_1 /\ B => E_1#) \in \Theta.
      \theta' \in \Theta'_n
         By Write Confinement, and R_0,
        we infer that there exists \phi_2 such that
        (\phi_1 /\ B) \/ (\phi_2 /\ ~B) => E_1# \in \Theta.
    Since s |= \phi_1 /\ B,  s1 |= \phi_1 /\ B
      and since s&s1 |=\beta \Theta,
     we in both cases infer the desired [[E_1]]s \beta [[E_1]]s1.

   Having established s&s1 |=\beta \Theta_1, we can apply the
    induction hypothesis on TS1, to find \beta' extending \beta such that
    s'&s'1 |=\beta' \Theta' and h'&h'1 |=\beta' I.

 * [[B]]s = false, [[B]]s1 = true
   Then s,h [[TS2]] s',h' and  s1,h1 [[TS1]] s'1,h'1.
   We shall now prove the claim with \beta' = \beta, that is,
    show that s'&s'1 |=\beta \Theta' and h'&h'1 |=\beta I.
  First we shall show:
    (*) there cannot be any \theta' \in \Theta'_m with
```

```
        s' |= \lhs(\theta'), s'1 |= lhs(\theta').
  For assume there exists such \theta'.
     Then, by Lemma BackSatExists, there exists
          (\phi_1 => E_1#,_,\theta') \in R_1
      and (\phi_2 => E_2#,_,\theta') \in R_2 such that
       s |= \phi_2, s1 |= \phi_1.
    But then, by construction, we would have \phi => B# \in \Theta
      with \phi = \phi_1 /\ B \/ \phi_2 /\ ~B,
      so since s |= \phi_2 /\ ~B and hence s |= \phi,
      and s1 |= \phi_1 /\ B and hence s1 |= \phi,
     we would have [[B]]s \beta [[B]]s1, yielding a contradiction.

  Next we shall show s'&s'1 |=\beta \Theta',
    so let \theta' = \phi' => E# \in \Theta',
    and assume that s' |= \phi', s'1 |= \phi'
      so as to prove [[E]]s' \beta [[E]]s'1.
   From (*), we infer that \theta' \in \Theta'_u.
   By Write Confinement on S, there exists unique
          (\phi => E#,\_,\theta') \in R
   and s,s' agree on fv(E), s1,s'1 agree on fv(E).
    By Lemma BackSatExists we infer
        s |= \phi, s1 |= \phi, so from s&s1 |=\beta \Theta
    we infer [[E]]s \beta [[E]]s1. But then also
      [[E]]s' \beta [[E]]s'1, as desired.

 Finally, we shall show h'&h'1 |=\beta I,
   so consider l,l1 with l \beta l1.
    Here, l \in dom(h), and l1 in dom(h1).
   With r' = h'(l) and r = h(l) and r'1 = h'1(l1) = r1 = h1(l1),
    we must prove that r'&r'1 |=\beta I, given that r&r1 |=\beta I.
    So given \phi_0 => E_0# \in I, and assume that r' |= \phi_0
     and r'1 |= \phi_0, we must show [[E_0]]r' \beta [[E_0]]r'1.
   By our overall assumption, there exists \theta'_0 \in \Theta' with
       s' |= \lhs(\theta'_0), s'1 |= \lhs(\theta'_0).
     By (*), we infer that \theta'_0 \in \Theta_u,
    and thus R contains no (_,m,\theta'_0).
     By Lemma~\ref{lem:wc2} on S, we now infer that
        r |= \phi_0 and that for all f in ff(E_0), r(f) = r'(f).
        r1 |= \phi_0 and that for all f in ff(E_0), r1(f) = r'1(f).
    From r&r1 |=\beta I, we infer [[E_0]]r \beta [[E_0]]r1,
       and hence the desired [[E_0]]r' \beta [[E_0]]r'1.

TS = new x in RS close
  Assume that s,h [[S]] s',h', and s1,h1 [[S]] s'1,h'1.
  Thus there exists l notin dom(h),ran(h),ran(s),
```

```
   there exists l1 notin dom(h1),ran(h1),ran(s1),
  such that with r = default we have
      s{x->l},r |= [[RS]] s',r'    s1{x->l1},r |= [[RS]] s'1,r'1
   h' = h{l -> r'}, h'1 = h1{l1 -> r'1}

 We now define \beta' as \beta \cup {(l,l1)}.
 We assume that s&s1 |=\beta \Theta and h&h1 |=\beta  I,
  and must prove s'&s'1 |=\beta' \Theta' and h'&h'1 |=\beta' I.

 First let us prove s{x->l},r&s1{x->l1},r |=\beta' \Theta_0.
 So let \theta_0 \in \Theta_0 with \theta_0 = \phi_0 => E_0#,
   and assume that s{x->l},r |= \phi_0, s1{x->l1},r |= \phi_0
   so as to prove [[E_0]]s{x->l},r \beta' [[E_0]]s1{x->l1},r.
   Note that with \phi = RemL_x(\phi_0)[f -> default] and
       E = RemR_x(E_0[f->default(f)])
    we have \phi => E# \in \Theta.
   By (repeated application of) Lemma PhiSub2, we infer that
     s{x->l} |= \phi_0{f -> def(f)}, s1{x->l1} |= \phi_0{f -> def{f}}
   From the Lemma on RemL_x we infer s |= \phi and s1 |= \phi.
   So from \phi => E# \in \Theta
      and s&s1 |=\beta \Theta we conclude that [[E]]s \beta [[E]]s1.
   Two cases:
    * x notin fv(E_0).
        Then E = E_0[f->default(f)],
      and by Lemma ExpSub2 we infer [[E_0]]s,r \beta [[E_0]]s1,r
        which clearly implies the desired
            [[E_0]]s{x->l},r \beta' [[E_0]]s1{x->l1},r.
    * x in fv(E_0).
       Then the Lemma about RemR_x tells us that
           [[E_0]]s{x->l},r \beta' [[E_0]]s1{x->l1},r.

Having proved s{x->l},r&s1{x->l1},r |=\beta' \Theta_0,
 the correctness result for RS tells us that
   s',r'&s'1,r'1 |=\beta' \Theta' \cup I.
 In particular, we have s'&s'1 |=\beta' \Theta',
  and r'&r'1 |=\beta' I.

 Next let us prove h'&h'1 |=\beta' I. That is, for all
     l',l'1 with l' \beta' l'1 it must hold that
     h'(l')&h'1(l'1) |=\beta' I.
 If l' = l and thus l'1 = l1, this results amounts
   to r'&r'1 |= \beta' I which we have just proved.
 Otherwise, we have l' \beta l'1 with l' \in dom(h), l'1 \in dom(h1).
     By assumption, we have h(l')&h1(l'1) |=\beta I.
      But since h'(l') = h(l') and h'1(l'1) = h1(l'1),
```

```
     this amounts to the desired h'(l')&h'1(l'1) |=\beta' I.

TS = open x in RS close
  Assume that s,h [[TS]] s',h' because with l = s(x) and r = h(l)
   we have h' = h{x -> r'} where s,r [[RS]] s',r';
  and assume that s1,h1 [[S]] s'1,h'1 because with l1 = s1(x) and r1 = h1(l1)
   we have h'1 = h1{x -> r'1} where s1,r1 [[RS]] s'1,r'1.
  Assume that s&s1 |=\beta \Theta and h&h1 |=\beta  I.
  We shall now prove the claim with \beta' =  \beta, that is, show that
    s'&s'1 |=\beta \Theta' and h'&h'1 |=\beta I.
 Two cases:
  * it does not hold that l \beta l1.
     We shall first show s'&s'1 |=\beta \Theta',
      so consider \theta' = \phi' => E'# \in \Theta',
      and assume that s' |= \phi', s'1 |= \phi',
        so as to show [[E']]s' \beta [[E']]s'1.
    By BackSatExists applied to RS, there exists
       \theta = \phi => E# such that (\theta,g,\theta') \in R_0
       and such that s,r |= \phi, s1,r1 |= \phi.
     If g = m, we get a contradiction as follows:
       from R1 we infer that RemF+(\phi) => x# \in \Theta.
       Note that s |= RemF+(\phi), s1 |= RemF+(\phi),
      so from s&s1 |=\beta \Theta
        we infer s(x) \beta s1(x), that is l \beta l1, a contradiction.
     Thus g = u, and from Write Confinement on RS we infer
        E' = E, and that s,s' agree on fv(E), and that s1,s'1 agree on fv(E).
     From R4 we see that Rem+(\phi) => E# \in \Theta,
        so since s |= Rem+(\phi) and s1 |= Rem+(\phi) and s&s1 |=\beta \Theta
      we infer [[E]]s \beta [[E]]s1, that is,
         the desired [[E']]s' \beta [[E']]s'1.

    Next we shall show h'&h'1 |=\beta I,
     so consider l' \beta l'1. Our assumptions are that
           h(l')&h1(l'1) |=\beta I
     and we must prove
           h'(l')&h'1(l'1) |=\beta I.
    This is obvious if l' \neq l, l'1 \neq l1 as then
        h'(l') = h(l'), h'1(l'1) = h1(l'1).
    So assume, wlog, that l' = l. Since l' \beta l'1 and not(l \beta l1),
        we infer that l1 \neq l'1.
     With r0 = h1(l'1) = h'1(l'1),
       we must prove r'&r0 |=\beta I, given r&r0 |=\beta I.
     Consider now \phi_0 => E_0# \in I, and thus r&r0 |=\beta \phi_0 => E_0#,
      and assume r' |= \phi_0 and r0 |= \phi_0,
          we must prove [[E]]r' \beta [[E]]r0.
```

```
   By Totality on RS,
      there exists (\phi=>E#,g,\phi_0 => E_0#) \in R_0
    and here g = u, for otherwise we could infer from R2 that
      true => x# \in \Theta and hence (since s&s1 |=\beta \Theta)
      s(x) \beta s1(x) which is a contradiction.
   Write Confinement and BackSatExists on RS now tells us that E = E_0,
     and that r’ and r agree on fv(E_0),
     and that s,r |= \phi. Since LogImp1(\phi,\phi_0) \in VC_1 \subseteq VC
      we infer from |= VC that r |= \phi_0.
   Since r0 |= \phi_0, we infer that [[E_0]]r \beta [[E_0]]r0,
    and hence the desired [[E_0]]r’ \beta [[E_0]]r0.

* l \beta l1.
  We shall first show that s,r&s1,r1 |=\beta \Theta_0.
    When that is in place, correctness of RS tells us that
    s’,r’&s’1,r’1 |=\beta \Theta’ \cup I, implying
    s’&s’1 |=\beta \Theta’ and r’&r’1 |=\beta I.
   This is as desired, except we must show that
     h’(l’)&h’1(l’1) |=\beta I  for l’ \beta l’1.
    But if l’ = l, then l’1 = l1 and the claim follows from r’&r’1 |=\beta I.
    And if l’ \neq l, then l’1 \neq l1 and the claim amounts to
    h(l)&h1(l’1) |=\beta I which follows from our assumption h&h1 |=\beta I.

 We now embark on proving s,r&s1,r1 |=\beta \Theta_0,
    given s&s1 |=\beta \Theta and r&r1 |=\beta I.
  So let \theta_0 \in \Theta_0, with \theta_0 = \phi_0 => E_0#,
      we shall prove s,r&s1,r1 |=\beta \phi_0 => E_0#.
   Note that it is sufficient to prove
     s,r&s1,r1 |=\beta RemF+(\phi_0) => E_0#.

  By Totality, there exists \theta’ \in \Theta’ \cup I such that
       (\theta_0,g,\theta’) \in R_0.
 Two cases:
   * g = m. Two subcases:
      * If E_0 is field-free, then from R3 we see that
            RemF+(\phi_0) => E_0# in \Theta.
          Since s&s1 |=\beta \Theta,
           this clearly yields the claim.
      * If E_0 is not field-free, then
         LogImp2(I,\phi_0 => E_0) \in VC_2 \subsetet VC
          so from |= VC and r&r1 |=\beta I we infer
          the desired s,r&s1,r1 |=\beta \phi_0 => E_0#.
  * g = u. Two subcases:
     *  If \theta’ \in \Theta’, then from R4 we see that
            RemF+(\phi_0) => E_0# in \Theta.
```

```
          Since s&s1 |=\beta \Theta,
           this clearly yields the claim.
     *  If  \theta' \in I, then with \theta' = \phi' => E'# we have
          LogImp1{\phi_0}{\phi'} \in VC_1 \subseteq VC
         so from |= VC we infer that \phi_0 logically implies phi'.
        From WriteConfinement we know that E' = E.
        From r&r1 |=\beta I we have r&r1 |=\beta \phi' => E#.
        But then we can infer the desired s,r&s1,r1 |=\beta \phi_0 => E#.

S = while B do S0.
It is clearly sufficient to prove the following result:
 Assume that s&s1 |=\beta \Theta and h&h1 |=\beta I.
 Assume that there exists i,j such that
    s,h f_i s',h'   and s1,h1 f_j s'1,h'1.
 Further assume that there exists \theta_0 \in \Theta such that
    s' |= \lhs(\theta_0), s'1 |= \lhs(\theta_0).
 Then there exists \beta' extending \beta such that
      s'&s'1 |=\beta' \Theta'_0 and h'&h'1 |=\beta' I.
Proof: We shall proceed by induction in i+j.
 Apart from symmetry, there are three cases:
* i = j = 0: then the claim is obvious, with \beta' = \beta,
    as s' = s, h' = h, s'1 = s1, h'1 = h1.
* i > 0, j > 0.
   Here s |= B, s1 |= B,
    and there exists s'',h'', s''1,h''1 such that
     s,h [[S_0]] s'',h'', s1,h1 [[S_0]] s''1,h''1
    s'',h''  f_{i-1} s',h'   s''1,h''1 f_{j-1} s'1,h'1.
  First observe that s&s1 |=\beta \Theta_0.
    For assume that \phi_0 => E_0# \in \Theta_0,
      and that s |= \phi_0 and s1 |= \phi_0,
       so as to prove [[E_0]]s \beta [[E_0]]s1.
   Since LogImp2(\Theta, \phi_0 /\ B => E_0#) in VC1 \subseteq VC
    we from |= VC infer s&s1 |=\beta \phi_0 /\ B => E_0#.
      So from s |= \phi_0 /\ B and s1 |= \phi_0 /\ B
      we infer the desired [[E_0]]s \beta [[E_0]]s1.

  By Lemma BackSatExists, applied to \theta_0, we next infer that there exists
   \theta_1 such that if s'',h'' [[while B do S0]] s',h' and
       s''1,h''1 [[while B do S0]] s'1,h'1 then
      s'' |= \lhs(theta_1) and s''1 |= \lhs(theta_1).
  Therefore we can apply the outermost induction hypothesis on S0,
  so as to find \beta'' extending \beta such that
      s''&s''1 |=\beta'' \Theta'_0, h''&h''1 |=\beta'' I
  We can now use the innermost induction hypothesis to find
     \beta' extending \beta'' such that
```

```
      s'&s'1 |=\beta' \Theta'_0, h'&h'1 |=\beta' I.
  This is as desired, since \beta' extends \beta''.

* i > 0, j = 0.
  Then [[B]]s = true, and [[B]]s1 = false, so s'1 = s1.
    We shall show the claim with \beta' = \beta.
  First observe that

     if \theta = \phi => E# is such that
     s' |= \phi and s1 |= \phi then \theta \notin \Theta_m.

For assume that \theta \in \Theta_m, so as to get a contradiction.
    By Lemma BackSatExists applied to S
      we infer s |= \lhs(\theta_m) and s1 |= \lhs(\theta_m).
    Since LogImp2(\Theta, lhs(\theta_m) => B#) \in VC2 \subseteq VC
     we infer from |= VC and s&s1 |= \Theta that
          s&s1 |= \lhs(\theta_m) => B#, and thus
       [[B]]s \beta [[B]]s1. But as we cannot have true \beta false,
       this is a contradiction.

 We shall first show s'&s1 |=\beta \Theta, so consider
 \theta = \phi => E# \in \Theta, and assume that s' |= \phi, s1 |= \phi,
   so as to show [[E]]s' \beta [[E]]s1.
 From the above observation we infer that \theta \in \Theta_u.
 Lemma BackSatExists, applied to S, then tells us that s |= \phi.
 Since there exists no (_,m,\theta) \in R_0,
 Lemma Write Confinement will tell us that
     s,s' agree on fv(E).
 Since s&s1 |= \phi => E#, we from s |= \phi and s1 |=\phi infer
   [[E]]s \beta [[E]]s1, and thus the desired [[E]]s' \beta [[E]]s'1.

 Finally, we shall show h'&h1 |=\beta I.
So consider \phi_0 => E_0# \in I, let l \beta l1,
   let r = h(l), r' = h'(l), r1 = h1(l).
  We must prove r'&r1 |= \phi_0 => E_0#, so assume
     r' |= \phi_0, r1 |= \phi_0 so as to prove [[E_0]]r' \beta [[E_0]]r1.
 Recall that there exists \theta'_0 \in \Theta such that
    s' |= \lhs(\theta'_0), s'1 |= \lhs(\theta'_0).
 From the above observation we infer that \theta'0 in R_u.
 Thus, there exists no (_,m,\theta'_0) \in R_0,
  so by Lemma~\ref{lem:wc2} applied to f_i, we infer that
  * r |= \phi_0, so from r&r1 |= I we infer [[E_0]]r \beta [[E_0]]r1;
  * for all f in ff(E_0), r(f) = r'(f), so we infer the
     desired [[E_0]]r' \beta [[E_0]]r1.
```

□

B.6 Material to be inserted

Semantics. Remaining clauses:

```
  s,r  [[skip]]  s'.r'
     iff s' = s, r' = r

  s,r  [[x := A]] s',r'
     iff there exists v such that
       v = [[A]]s,r
       s' = s{x -> v}, r' = r

   s,r [[if B then RS1 else RS2]] s',r'
        iff
     [[B]]s,r = true  implies s,r [[RS1]] s',r'
     [[B]]s,r = false implies s,r [[RS2]] s',r'

   s,r [[while B then RS]] s',r'
       iff exists i >= 0:
               s,r  f_i  s',r'
       where f_i is given recursively as follows:
            s,r   f_0   s',r' iff [[B]]s,r = false, s' = s, r' = r,
            s,r f_{i+1} s',r' iff
               exists s'',r'':
                     [[B]]s,r = true
                      s,r [[RS]] s'',r''
                      s'',r'' f_i s',r'

  s,h  [[skip]]  s',h'
     iff s' = s, h' = h

  s,h [[assert(\phi)]] s',h'
     iff s |= \phi,
        s' = s, h' = h

  s,h  [[S1; S2]] s',h'
     iff exists s'',h'':
      s,h [[S1]] s'',h'',
      s'',h'' [[S2]] s',h'
```

Facts about semantics.

```
If s,r [[RS]] s',r'
 then dom(s) \subseteq dom(s').
```

```
If s,h [[RS]] s',h'
 then dom(s) \subseteq dom(s').
 and  dom(h) \subseteq dom(h').
```

Simple worked out example

```
 Given the program
if pin = 1234
then out := x
else out := y
 and postcondition: pin = 1234 => out# /\ pin != 1234 => out#

We get the following assertions
(where assertions with the same "label" are connected by R)
with all arcs labeled "m"

1: pin = 1234 => x#
1: false => 0#
1: true => 0#
2: false => 0#
2: pin != 1234 => y#
2: true => 0#
  simplify
1: pin = 1234 /\ pin = 1234 => x#
1: pin = 1234 /\ pin != 1234 => y#
1: pin = 1234 /\ pin = 1234 \/ pin = 1234 /\ pin != 1234 => (pin = 1234)#
2: pin != 1234 /\ pin = 1234 => x#
2: pin != 1234 /\ pin != 1234 => y#
2: pin != 1234 /\ pin = 1234 \/ pin != 1234 /\ pin != 1234 => (pin = 1234)#
  if pin = 1234
1: pin =  1234 => x#
2: pin != 1234 => x#
 then out := x
1: pin =  1234 => y#
2: pin != 1234 => y#
 else out := y
{out}
1: pin = 1234 => out#
2: pin != 1234 => out#

that is we end up with the expected preconditions
 pin = 1234 => x#
 pin != 1234 => y#
as well as 4 which are always true.
```

Remark about simplification.

```
Notice: We might have wanted to allow for say
   \phi => (x+y)# to simplify to
     \phi => x#, \phi => y#
  through a connection tagged u.
 This would be OK, as long as we don't throw away free variables,
   and as long as x# and y# implies (x+y)#,
  but it will make the statement of Write Confinement more complex.
 On the other hand, if the connection is to be tagged ``u'',
   we can not allow the apparently innocent simplification of
          x = 3 => (x-3)#   into   true => 0#
Counterexample:
   if h > 8
   then
      x := r
      simplify;
      z := x
   else
      y := q;
      simplify;
      z := y
   fi
{z = 3 => (x+y)#}
Doing naive optimization, we may get, with all arcs labeled by u
{r = 3 /\ h > 8 \/ q = 3 /\ h <= 8 => y#}
{r = 3 /\ h > 8 \/ q = 3 /\ h <= 8 => x#}
   if h > 8
   then
{r = 3 => y#}
      x := r
{x = 3 => y#}
      simplify
{x = 3 => (x+y)#}
      z := x
   else
{q = 3 => x#}
      y := q;
{y = 3 => x#}
      simplify
{y = 3 => (x+y)#}
      z := y
   fi
{z = 3 => (x+y)#}

But the pre-two-state
```

```
   h q r x y z       h q r x y z
   9   3 8 5         7 3   8 5
satisfies the precondition,
whereas the corresponding post-two-state
   h'q'r'x'y'z'      h'q'r'x'y'z'
         3 5 3             8 3 3
does *not* satisfy the postcondition!
```

A more refined version of ff^+**.**

```
We shall define RemF+ simultaneously with its dual RemF-(\phi)
which has the property that with \phi' = RemF-(\phi) we have
  * \phi' does not contain any field names
  * For all s,r: if s |= \phi' then s,r |= \phi
RemF+(B) = if B contains field names then true else B
RemF+(\phi_1 or \phi_2) = RemF+(\phi_1) or RemF+(\phi_2)
RemF+(\phi_1 and \phi_2) = RemF+(\phi_1) and RemF+(\phi_2)
RemF+(~\phi) = ~RemF-(\phi)
RemF-(B) = if B contains field names then false else B
RemF-(\phi_1 or \phi_2) = RemF-(\phi_1) or RemF-(\phi_2)
RemF-(\phi_1 and \phi_2) = RemF-(\phi_1) and RemF-(\phi_2)
RemF-(~\phi) = ~RemF+(\phi)
```

Simplification. We must argue that the proposed simplifications obey all of the correctness results: Lemmas 4.3,4.5,4.6; Proposition 4.1, Theorem 4.2.

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

$[VC]\{\Theta\}\ (R) \Longleftarrow \textbf{assert}(\phi_0)\ \{\Theta'\}$
  iff $R = \{((\phi \wedge \phi_0) \Rightarrow E\ltimes, u, \phi \Rightarrow E\ltimes) \mid \phi \Rightarrow E\ltimes \in \Theta'\}$
  and $\Theta = dom(R)$ and $VC = \emptyset$

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

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

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

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

Figure 5: The verification condition generator, part I

$[VC]\{\Theta\}\ (R) \Longleftarrow \textbf{while}\ B\ \textbf{do}\ S_0\ \{\Theta\}$
  iff $[VC_0]\{\Theta_0\}\ (R_0) \Longleftarrow S_0\ \{\Theta\}$
  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 $VC_1 = \{\Theta \rhd^2 (\phi \wedge B) \Rightarrow E\ltimes \mid (\phi \Rightarrow E\ltimes, \_, \_) \in R_0\}$
    and $VC_2 = \{\Theta \rhd^2 \phi_m \Rightarrow B\ltimes\}$
    and $VC_3 = \{ant(\theta) \rhd^1 \phi_m \mid \theta \in \Theta_m\}$
    and $VC_4 = \{ant(\theta) \rhd^1 \phi_m \mid (\theta, \_, \theta_m) \in R_0\}$
    and $VC_5 = \{ant(\theta_0) \rhd^1 ant(\theta) \mid (\theta_0, \_, \theta) \in R_0,\ \theta \in \Theta_u\}$
    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 $\phi_m = ant(\theta_m)$

$[VC]\{\Theta\}\ (R) \Longleftarrow \textbf{open}\ x\ \textbf{in}\ RS\ \textbf{close}\ \{\Theta'\}$
  iff $[VC_0]\{\Theta_0\}\ (R_0) \Longleftarrow RS\ \{\Theta' \cup \mathcal{I}\}$
  and $R = R_1 \cup R_2 \cup R_3 \cup R_4$
    where $R_1 = \{(\mathit{ff}^+(\phi) \Rightarrow x\ltimes, m, \theta') \mid \theta' \in \Theta', (\phi \Rightarrow \_\ltimes, m, \theta') \in R_0\}$
    and $R_2 =$ if exists $\theta \in \mathcal{I}$ with $(\_, m, \theta) \in R_0$ then $\{(true \Rightarrow x\ltimes, m, \theta') \mid \theta' \in \Theta'\}$ else $\emptyset$
    and $R_3 = \{(\mathit{ff}^+(\phi) \Rightarrow E\ltimes, m, \theta') \mid \theta' \in \Theta',\ E$ field-free, $\exists\theta_0' \in \mathcal{I} \cup \{\theta'\} \bullet (\phi \Rightarrow E\ltimes, m, \theta_0') \in R_0\}$
    and $R_4 = \{(\mathit{ff}^+(\phi) \Rightarrow E\ltimes, u, \theta') \mid \theta' \in \Theta',\ (\phi \Rightarrow E\ltimes, u, \theta') \in R_0\}$
  and $\Theta = dom(R)$ and $VC = VC_0 \cup VC_1 \cup VC_2$
    where $VC_1 = \{ant(\theta) \rhd^1 ant(\theta') \mid \theta' \in \mathcal{I},\ (\theta, u, \theta') \in R_0\}$
    and $VC_2 = \{\mathcal{I} \rhd^2 \theta \mid (\theta, m, \_) \in R_0,\ con(\theta)$ not field-free$\}$

$[VC]\{\Theta\}\ (R) \Longleftarrow \textbf{new}\ x\ \textbf{in}\ RS\ \textbf{close}\ \{\Theta'\}$
  iff $[VC_0]\{\Theta_0\}\ (R_0) \Longleftarrow RS\ \{\Theta' \cup \mathcal{I}\}$
  and $R = \{(rm_x(\phi[\overline{deflt(f)}/\overline{f}]) \Rightarrow rm_x(E[\overline{deflt(f)}/\overline{f}])\ltimes, \gamma, \theta')$
    $\mid (\phi \Rightarrow E\ltimes, \gamma_0, \theta') \in R_0,\ \gamma = m$ iff $\gamma_0 = m$ or $x \in \mathrm{fv}(E)\}$
  and $\Theta = dom(R)$ and $VC = VC_0$

Figure 6: The verification condition generator, part II

$\{true \Rrightarrow x\ltimes\}$

1. **open** x **in**

$\{odd(.idx) \Rrightarrow .src\ltimes,\ true \Rrightarrow x\ltimes,\ odd(.idx) \Rrightarrow .val\ltimes\}$

2. $\quad y := .src;$

//Case of field access: replace y by $.src$ to obtain pre

$\{odd(.idx) \Rrightarrow y\ltimes,\ true \Rrightarrow x\ltimes,\ odd(.idx) \Rrightarrow .val\ltimes,\ odd(.idx) \Rrightarrow .src\ltimes\}$

3. $\quad i := .idx;$

//Case of field access: replace i by $.idx$ to obtain pre

$\{odd(i) \Rrightarrow y\ltimes,\ true \Rrightarrow x\ltimes,\ odd(.idx) \Rrightarrow .val\ltimes,\ odd(.idx) \Rrightarrow .src\ltimes\}$

// Conjoin object invariant to post

4. **close**;

$\{odd(i) \Rrightarrow y\ltimes,\ true \Rrightarrow x\ltimes\}$

5. **open** y **in**

$\{(odd(i) \rightarrow odd(.idx)) \Rrightarrow x\ltimes,$
$odd(i) \wedge (odd(i) \rightarrow odd(.idx)) \Rrightarrow .val\ltimes,$
$odd(.idx) \wedge (odd(i) \rightarrow odd(.idx)) \Rrightarrow .val\ltimes,$
$odd(.idx) \wedge (odd(i) \rightarrow odd(.idx)) \Rrightarrow .src\ltimes\}$

6. $\quad$ **assert** $(odd(i) \rightarrow odd(.idx));$

//Conjoin assertion to obtain pre

$\{true \Rrightarrow x\ltimes,\ odd(i) \Rrightarrow .val\ltimes,\ odd(.idx) \Rrightarrow .val\ltimes,\ odd(.idx) \Rrightarrow .src\ltimes\}$

7. $\quad q := .val;$

//Case of field access: replace q by $.val$ to obtain pre

$\{true \Rrightarrow x\ltimes,\ odd(i) \Rrightarrow q\ltimes,\ odd(.idx) \Rrightarrow .val\ltimes,\ odd(.idx) \Rrightarrow .src\ltimes\}$

// Conjoin object invariant to simplified post

8. **close**;

$\{odd(i) \Rrightarrow x\ltimes,\ true \Rrightarrow x\ltimes,\ odd(i) \Rrightarrow q\ltimes\}$

9. **open** x **in**

$\{odd(i) \wedge (.idx = i) \Rrightarrow q\ltimes, odd(.idx) \wedge (.idx = i) \Rrightarrow q\ltimes,$
$odd(.idx) \wedge (.idx = i) \Rrightarrow .src\ltimes\}$

10. $\quad$ **assert** $(.idx = i);$

//Conjoin assertion to obtain pre

$\{odd(i) \Rrightarrow q\ltimes, odd(.idx) \Rrightarrow q\ltimes, odd(.idx) \Rrightarrow .src\ltimes\}$

11. $\quad .val := q;$

//Case of field update: replace $.val$ by q to obtain pre

$\{odd(i) \Rrightarrow .val\ltimes, odd(.idx) \Rrightarrow .val\ltimes, odd(.idx) \Rrightarrow .src\ltimes\}$

12. $\quad result := .val;$

//Case of field access: replace $result$ by $.val$ to obtain pre

$\{odd(i) \Rrightarrow result\ltimes, odd(.idx) \Rrightarrow .val\ltimes, odd(.idx) \Rrightarrow .src\ltimes\}$

// Conjoin object invariant to post

13. **close**;

$\{odd(i) \Rrightarrow result\ltimes\}$

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