Abstract Interpretation of Symbolic Execution with Explicit State Updates*

Richard Bubel[1], Reiner Hähnle[1], and Benjamin Weiß[2]

[1] Department of Computer Science and Engineering,
Chalmers University of Technology and Göteborg University
{bubel,reiner}@chalmers.se
[2] Institute for Theoretical Computer Science,
University of Karlsruhe
bweiss@ira.uka.de

Abstract. Systems for deductive software verification model the semantics of their target programming language with full precision. On the other hand, abstraction based approaches work with approximations of the semantics in order to be fully automatic. In this paper we aim at providing a uniform framework for both fully precise and approximate reasoning about programs. We present a sound dynamic logic calculus that integrates abstraction in the sense of abstract interpretation theory. In the second part of the paper, we apply the approach to the analysis of secure information flow.

1 Introduction

Formal verification of software is desirable for many safety- and security-critical applications. Following intense research during the last decade, the reach of formal verification methods has been extended impressively. Different approaches to verification are often categorized as "interactive" versus "automatic" depending on whether they in general require hints from human users or not. Typical interactive systems include generic proof assistants and logical frameworks such as Isabelle [20] as well as deductive verification systems such as KeY [5], KIV [2], Spec# [3], and Why/Krakatoa/Caduceus [10].[1] Typical automated systems include model checkers such as Bogor [21], Java PathFinder [25], Spin [15] and abstract interpreters such as ASTRÉE [8].

Deductive verification systems model the semantics of their target programming language with full precision. This is the source of the need for user interaction, because all interesting properties of Turing-complete programming

* This work was funded in part by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project. This article reflects only the authors' views and the Community is not liable for any use that may be made of the information contained therein.

[1] We classify systems based on a verification condition generator architecture such as Spec# or Why as interactive, because in general human users have to enrich specifications incrementally until they can be proven.

F.S. de Boer et al. (Eds.): FMCO 2008, LNCS 5751, pp. 247–277, 2009.

languages are undecidable. Technically, the necessity of user interaction arises when suitable invariants or induction hypotheses are required that characterize the effect of unbounded loops or recursion.

Automatic verification approaches avoid interaction by working on abstract execution models (and specification languages) that are decidable or even have a finite state space. This allows the full exploration of the state space of a program (as in model checking) or finite fixed point approximation of invariants (as in abstract interpretation).

There are various attempts to combine the advantages of verification systems and abstraction-based approaches, usually, by using the latter to boost the degree of automation of the former (for example, [18,24]). In the present paper we set a more ambitious goal: we want to provide a uniform theoretical basis for fully precise reasoning about programs *and* for abstract interpretation at the same time. The aim is to achieve a deep integration of deductive verification and abstract interpretation. One obvious reason is to be able to re-use the substantial investments and progress made in the context of deductive verification in the past years to improve the precision of abstract interpretation. Another important motivation for this work is the possibility to achieve automation of deductive verification without completely losing precision.

Our starting point is a program logic that allows to cast symbolic program execution as deduction in a sequent calculus. This so-called *dynamic logic* is an extension of first-order logic and is complete relative to arithmetic. The software verification systems KeY [5] and KIV [2] formally model large fragments of the JAVA programming language based on dynamic logic. Our exposition in Sect. 2 is based on a simplified[2] version of the KeY logic [5, Chapter 3].

Section 3 is the core of the paper: we define a calculus for logic-based symbolic execution that allows for any program variable at any time to move from concrete symbolic execution to computation in an abstract domain. The abstract domain is a sound approximation of the program in the sense of abstract interpretation theory [7]. The approach is based on the symbolic state *updates* featured by our logic-based symbolic execution: a very compact language for representing the intermediate results of symbolic computation. It is on these updates that abstraction takes place, not on full target programs. While it is still possible to use the calculus interactively and let the user specify the loop invariants, the abstraction also makes an automatic procedure possible, where loop invariants are derived without interaction by iterating symbolic execution of the loop body until stabilization to a fixed point. The overall approach is illustrated by an extended example in Sect. 4.

One potentially very rewarding area for a program logic with abstraction such as suggested here is the analysis of secure information flow. This problem has received a lot of attention in the past years with many type-based (see [23] for an overview), some deduction-based (for example, [17,4,1,9,6,13]) and a few

[2] Ultimately, we aim to cover as much of JAVA as done in the KeY system based on the logic described in this paper, but in order to stay reasonably short and comprehensible we give formal definitions only for a toy programming language.

abstract interpretation-based approaches (for example, [11]). The information flow analysis problem has also been the original motivation for the work undertaken here.

While type-based approaches to information flow analysis are automatic, but suffer from limited precision, most deduction-based approaches recast flow analysis as a general verification problem [4,9,6] that typically requires user interaction to prove it. Other deduction-based approaches provide a logical model of type-based flow analysis [1,13], but this results in rather specialized calculi with limited prospects of re-use of existing verification systems.

In the second part of this paper we extend our symbolic execution/abstract interpretation framework to model secure information flow. It was shown by Hunt & Sands [16] that information flow policies can be expressed as mappings from a program variable to all those locations that may influence its value. This property was exploited in [13] where the symbolic execution machinery and update mechanism of a dynamic logic was used to keep track of the locations that a program variable depends on. By virtue of a simple abstraction rule from a certain point onwards during symbolic execution a program variable `x` could be made to record dependencies on other variables instead of precise values. Unfortunately, this meant that at this point all information on the symbolic value of `x` was discarded. It also lead to some non-standard and non-deterministic rules. In the present paper we avoid these disadvantages. In Sect. 5 we extend the semantics of our programming language such that the dependencies of the program variables are tracked explicitly. We give sound modifications of the affected symbolic execution rules with respect to this semantics.

In Sect. 6 we discuss additional related work not mentioned above. In Sect. 7 we give directions for future work and summarize our results.

2 A Dynamic Logic with Updates

In this section, we describe our logic for reasoning about programs. It is a simplified version of the dynamic logic of KeY [5, Chapter 3]. Compared to classical dynamic logic [14] its most important new feature is a new syntactic category called *updates* [22]. Updates are used to describe state changes in an explicit and programming language independent way. Our overview begins with *syntax* in Sect. 2.1, continues with *semantics* in Sect. 2.2, and ends with the *calculus* used for symbolic program execution in Sect. 2.3.

2.1 Syntax

The syntax is based on a (first-order) *signature*:

Definition 1 (Signature). *A* signature *is a tuple* $\Sigma = (\mathcal{F}, \mathcal{P}, \mathcal{PV}, \mathcal{V})$*, where* $\mathcal{F}$ *is a set of* function symbols*,* $\mathcal{P}$ *is a set of* predicate symbols*,* $\mathcal{PV}$ *is a finite set of* program variables*, and where* $\mathcal{V}$ *is a set of (logical)* variables*.*

Function and predicate symbols have fixed arities. We require that $\mathcal{F}$ *contains infinitely many function symbols of each arity.*

Note that program variables (i.e., variables occurring in programs) and logical variables (i.e., variables that may be quantified over) are separate syntactic categories. For the rest of this paper, we assume a fixed signature Σ. For this reason we drop the signature as a parameter in all subsequent definitions.

Definition 2 (Syntax). Terms t, formulas φ, updates $\mathcal{U}$ *and* programs $\mathtt{p}$ *are defined by the following grammar, where* $f \in \mathcal{F}$ *ranges over function symbols,* $p \in \mathcal{P}$ *over predicate symbols,* $\mathtt{x} \in \mathcal{PV}$ *over program variables, and* $y \in \mathcal{V}$ *over logical variables:*

$$
\begin{aligned}
t &::= f(t,\ldots,t) \mid \mathtt{x} \mid y \mid \mathit{if}(\varphi)\mathit{then}(t)\mathit{else}(t) \mid \{\mathcal{U}\}t \\
\varphi &::= \mathrm{true} \mid \mathrm{false} \mid p(t,\ldots,t) \mid \varphi \,\&\, \varphi \mid (\varphi \mid \varphi) \mid \varphi \mathrel{->} \varphi \mid !\varphi \mid \\
&\quad\ \forall y.\varphi \mid \exists y.\varphi \mid t \doteq t \mid \{\mathcal{U}\}\varphi \mid [\mathtt{p}]\varphi \\
\mathcal{U} &::= (\mathtt{x} := t \parallel \ldots \parallel \mathtt{x} := t) \\
\mathtt{p} &::= \mathtt{x = } t \mid \mathtt{p;p} \mid \mathtt{if\ (}\varphi\mathtt{)\ \{p\}\ else\ \{p\}} \mid \mathtt{while\ (}\varphi\mathtt{)\ \{p\}}
\end{aligned}
$$

Terms $f(t_1,\ldots,t_n)$ *and formulas* $p(t_1,\ldots,t_n)$ *must respect the arities of the symbols* f *and* p*, respectively. Terms and formulas that appear inside programs may not contain any logical variables, quantifiers, updates, or nested programs.*

An expression of the form $[\mathtt{p}]\varphi$ is a *program formula*. Intuitively, it denotes partial correctness of the program $\mathtt{p}$ with respect to the postcondition φ. The symbol $\doteq$ denotes referential equality. *Updates* are lists of pairs of locations (program variables) and terms. They are used to represent the incremental difference between two states within a computation. In the KeY system [5] updates render symbolic execution efficient. In the present paper updates provide a convenient layer between programs and logic where abstraction takes place.

We allow programs of the form `if (`φ`) {p}`, i.e., conditionals without an `else`-block. This can be seen as an abbreviation for `if (`φ`) {p} else {x = x}`, where $\mathtt{x} \in \mathcal{PV}$ is an arbitrary program variable.

Example 1. Let `p` denote the following program computing the Gaussian sum for the first `i` numbers and storing the result in `n`:

```
n = 0;
while (i>0) {
  i = i-1;
  n = n+i
}
```

We can state partial correctness of this program (with respect to a rather weak postcondition), for example, by $\mathtt{i} \geq 0 \mathrel{->} [\mathtt{p}](\mathtt{i} \doteq 0 \,\&\, \mathtt{n} \geq 0)$.

2.2 Semantics

The semantics of terms, formulas, updates and programs is based on an interpretation I of the function and predicate symbols, a *state* s giving values for the program variables, and a *variable assignment* β assigning values to the logical variables:

Definition 3 (Interpretations, States, Variable Assignments). *Given a universe* D *of values, an* interpretation I *is a function mapping every function symbol* $f \in \mathcal{F}$ *with arity* n *to a function* $I(f) : \mathsf{D}^n \to \mathsf{D}$ *and every predicate symbol* $p \in \mathcal{P}$ *with arity* n *to a relation* $I(p) \subseteq \mathsf{D}^n$*. A* state *is a function* $s : \mathcal{PV} \to \mathsf{D}$*; the set of all states is denoted* $\mathcal{S}$*. A* variable assignment *is a function* $\beta : \mathcal{V} \to \mathsf{D}$*.*

Definition 4 (Semantics). *Given a universe* D*, an interpretation* I*, a state* s *and a variable assignment* β*, we evaluate terms* t *to a value* $val_{I,s,\beta}(t) \in \mathsf{D}$*, formulas* φ *to a truth value* $val_{I,s,\beta}(\varphi) \in \{tt, ff\}$*, updates* $\mathcal{U}$ *to a result state* $val_{I,s,\beta}(\mathcal{U}) \in \mathcal{S}$*, and programs* $\mathtt{p}$ *to a set of states* $val_{I,s,\beta}(\mathtt{p}) \in 2^{\mathcal{S}}$*, where the cardinality of* $val_{I,s,\beta}(\mathtt{p})$ *is either* 0 *or* 1*. The evaluation function* $val_{I,s,\beta}$ *is formally defined in App. A.1.*

A formula φ *is called (logically)* valid *iff* $val_{I,s,\beta}(\varphi) = tt$ *for* all *interpretations* I*,* all *states* s *and* all *variable assignments* β*.*

For terms and formulas without updates and without programs, the evaluation $val_{I,s,\beta}$ is essentially defined as usual in first-order logic. For an *update* $\mathcal{U} = (\mathtt{x}_1 := t_1 \parallel \ldots \parallel \mathtt{x}_n := t_n)$, the result of $val_{I,s,\beta}(\mathcal{U})$ is the state which results from s by assigning the values of the terms t_i to the program variables $\mathtt{x}_i$ in parallel. In case of a clash between two sub-updates (i.e., when $\mathtt{x}_i = \mathtt{x}_j$ for $i \neq j$), the rightmost update "wins" and overwrites the effect of the other. The meaning of a term $\{\mathcal{U}\}t$ and of a formula $\{\mathcal{U}\}\varphi$ is that the result state of the update $\mathcal{U}$ should be used for evaluating t and φ, respectively.

A *program* is evaluated to the set of states that it may terminate in when started in s. We only consider deterministic programs, so this set is always either empty (if the program does not terminate) or it consists of exactly one state. The semantics of a program formula $[\mathtt{p}]\varphi$ is that φ should hold in all result states of the program $\mathtt{p}$, which corresponds to partial correctness of $\mathtt{p}$ wrt. φ.

2.3 Calculus

We reason about logical validity of dynamic logic formulas via a *sequent calculus*. A *sequent* is an expression of the form $\Gamma \Longrightarrow \Delta$, where Γ (called the *antecedent*) and Δ (called the *succedent*) are finite sets of formulas. The semantics of a sequent is defined as $val_{I,s,\beta}(\Gamma \Longrightarrow \Delta) = val_{I,s,\beta}(\bigwedge \Gamma \texttt{ -> } \bigvee \Delta)$. As usual, $\bigwedge \Gamma$ stands for the conjunction (&) and $\bigvee \Delta$ for the disjunction (|) of the formulas in Γ and in Δ, respectively (in an arbitrary order). A sequent calculus *rule* is an inference rule of the form

$$\frac{seq_1 \quad \ldots \quad seq_n}{seq}$$

where $seq_1, \ldots, seq_n$ (called the *premisses* of the rule) and seq (called the *conclusion* of the rule) are sequents. A rule is called *sound* iff logical validity of all the premisses implies logical validity of the conclusion.

A *proof tree* is constructed by starting with some root sequent to be proven, and then applying sequent rules. *Applying* a rule means to find a leaf in the proof

tree that is identical to the conclusion of a rule, and to add the rule's premisses as new children of the former leaf. Provided that all applied rules are sound, it is guaranteed that at any time during this process, validity of all the leaves implies validity of the root sequent. If one arrives at a tree whose leaves are all obviously valid, one has proven the validity of the original proof obligation.

To achieve finite representation of a calculus, sequent rules are denoted schematically. For example, the following schematic rule is applicable to all sequents where an arbitrary conjunctive formula $\varphi_1 \,\&\, \varphi_2$ occurs in the antecedent:

$$\mathsf{andLeft}\ \frac{\Gamma, \varphi_1, \varphi_2 \Longrightarrow \Delta}{\Gamma, \varphi_1 \,\&\, \varphi_2 \Longrightarrow \Delta}$$

We handle formulas with programs in them by transforming them into formulas without programs. This process can be understood as *symbolic execution* of the code: the rules walk through the program in a forward manner, at each step discharging the first statement, until the program has been dealt with completely. For example, an *assignment statement* is handled with the rule below:

$$\mathsf{assignment}\ \frac{\Gamma \Longrightarrow \{\mathcal{U}\}\{\mathtt{x} := t\}[\ldots]\varphi, \Delta}{\Gamma \Longrightarrow \{\mathcal{U}\}[\mathtt{x = } \ t\mathtt{;}\ \ldots]\varphi, \Delta}$$

The update $\mathcal{U}$ may have resulted from an assignment symbolically executed earlier. As a border case, this update may be empty and disappear. The notation "..." stands for an arbitrary "trail program" behind the assignment. As another border case, this trail program may be empty; then, the subformula $[\ldots]\varphi$ in the premiss is simply φ without a program attached to it.

The assignment rule transforms a program-level assignment into an equivalent update. This is a useful step because updates are in general easier to reason about than programs; for example, updates always terminate, and they never have implicit side effects. The difference between programs and updates becomes more profound when dealing with a more realistic programming language than the toy language considered in this paper, such as JAVA. In particular, updates are then helpful for a sound handling of the *aliasing* problem, without having to do case splits for every assignment [5, Chapter 3].

A *conditional statement* can be handled by splitting the proof depending on whether the guard is true or false:

$$\mathsf{ifElse}\ \frac{\Gamma, \{\mathcal{U}\}g \Longrightarrow \{\mathcal{U}\}[\mathtt{p1;}\ \ldots]\varphi, \Delta \qquad \Gamma, \{\mathcal{U}\}\,!g \Longrightarrow \{\mathcal{U}\}[\mathtt{p2;}\ \ldots]\varphi, \Delta}{\Gamma \Longrightarrow \{\mathcal{U}\}[\mathtt{if\ (}g\mathtt{)\ \{p1\}\ else\ \{p2\};}\ \ldots]\varphi, \Delta}$$

For a *loop*, the simplest approach is to *unwind* it:

$$\mathsf{loopUnwind}\ \frac{\begin{array}{l}\Gamma, \{\mathcal{U}\}g \Longrightarrow \{\mathcal{U}\}[\mathtt{p;\ while\ (}g\mathtt{)\ \{p\};}\ \ldots]\varphi, \Delta \\ \Gamma, \{\mathcal{U}\}\,!g \Longrightarrow \{\mathcal{U}\}[\ldots]\varphi, \Delta\end{array}}{\Gamma \Longrightarrow \{\mathcal{U}\}[\mathtt{while\ (}g\mathtt{)\ \{p\};}\ \ldots]\varphi, \Delta}$$

Obviously, unwinding is sufficient only if an upper bound on the number of loop iterations is known statically. In general, an *invariant* rule is needed. Unlike the

other rules described here, such a rule usually cannot be applied automatically, because it relies on the presence of a suitable loop invariant.

Updates can be simplified and applied to terms and formulas using the set of (schematic) rewrite rules provided in App. B.

Example 2. Suppose we want to prove the validity of this sequent:

$$\texttt{i} > 0 \Longrightarrow [\texttt{n=0; i=i-1; n=n+i}]\texttt{n} \geq 0$$

Applying the assignment rule two times gives us:

$$\texttt{i} > 0 \Longrightarrow \{\texttt{n} := 0\}\{\texttt{i} := \texttt{i} - 1\}[\texttt{n=n+i}]\texttt{n} \geq 0$$

Since the two updates are independent of each other, this can be rewritten to:

$$\texttt{i} > 0 \Longrightarrow \{\texttt{n} := 0 \parallel \texttt{i} := \texttt{i} - 1\}[\texttt{n=n+i}]\texttt{n} \geq 0$$

Another application of assignment and another round of update rewriting yields:

$$\texttt{i} > 0 \Longrightarrow \{\texttt{n} := 0 \parallel \texttt{i} := \texttt{i} - 1 \parallel \texttt{n} := 0 + \texttt{i} - 1\}\texttt{n} \geq 0$$

Note that now, the effect of the sub-update $\texttt{n} := 0$ is overwritten by the rightmost sub-update which also writes to n. Since the program has now been dealt with completely, we can syntactically apply the update to the postcondition $\texttt{n} \geq 0$ (also using the rewrite rules in App. B):

$$\texttt{i} > 0 \Longrightarrow 0 + \texttt{i} - 1 \geq 0$$

Proving this sequent is a matter of simple arithmetic reasoning.

3 A Dynamic Logic with Abstraction

The main motivation for incorporating abstraction into a symbolic execution framework is to achieve automation. The core issue is to discover loop invariants automatically instead of relying on a human user. Our main idea is to employ a fixed point algorithm that performs repeated symbolic executions of the loop body, interleaved with abstraction steps, until an invariant is found.

To this end, we first introduce a notion of an *abstract domain* in Sect. 3.1. We expect an abstract domain to be a lattice of "abstract values", each representing a set of possible concrete values. For every abstract value, we introduce a partially interpreted constant symbol into our logic. Partially interpreted in this context means that the interpretation of such a symbol can vary on the concrete value as long as the latter satisfies certain domain restrictions. These constant symbols are used to represent abstract values within our updates. During construction of a proof, abstraction can be performed as an instance of *logical weakening*, for which we define a sound rule in Sect 3.2.

The invariants found by our algorithm can be used to get rid of loops by using the loop invariant rule of Sect. 3.3. Since the invariants we derive are updates instead of formulas, this rule is slightly different from the classical loop invariant rule of dynamic logic. The algorithm itself is described in Sect. 3.4 and Sect. 3.5.

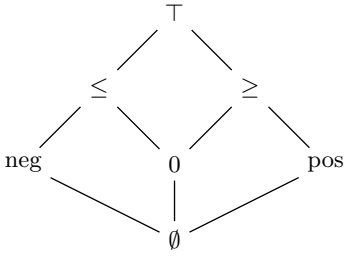
3.1 Abstract Domains

Definition 5 (Abstract Domains). *Given a universe* D *(which we will also call* concrete domain *from now on), an* abstract domain *is a countable lattice* $\mathcal{A}$ *with partial order* $\sqsubseteq$ *and join operator* $\sqcup$*. We require that* $\mathcal{A}$ *does not contain any infinite ascending chains. Further, an abstract domain comes with an* abstraction function $\alpha : 2^{\mathsf{D}} \to \mathcal{A}$ *and a* concretization function $\gamma : \mathcal{A} \to 2^{\mathsf{D}}$ *with the following properties (from [7]):*

1. α *and* γ *are monotone wrt. the partial orders* $\subseteq$ *and* $\sqsubseteq$
2. *for each* $a \in \mathcal{A} : a = \alpha(\gamma(a))$
3. *for each* $c \in 2^{\mathsf{D}} : c \subseteq \gamma(\alpha(c))$

The second property states that concretizing does not lead to a loss of information, while the third one expresses correctness of the abstraction: no concrete values are lost.

Example 3. As a simple example, our concrete domain may be $\mathsf{D} = \mathbb{Z}$, and our abstract domain $\mathcal{A}$ may be the sign lattice depicted in Fig. 1. The abstraction function α and concretization function γ are as usual for this domain. For convenience, γ is given in the right part of Fig. 1.



$$\begin{aligned}
\gamma(\top) &= \mathbb{Z} \\
\gamma(\leq) &= \{i \in \mathbb{Z} \mid i \leq 0\} \\
\gamma(\geq) &= \{i \in \mathbb{Z} \mid i \geq 0\} \\
\gamma(\text{neg}) &= \{i \in \mathbb{Z} \mid i < 0\} \\
\gamma(\text{pos}) &= \{i \in \mathbb{Z} \mid i > 0\} \\
\gamma(0) &= \{0\} \\
\gamma(\emptyset) &= \{\}
\end{aligned}$$

Fig. 1. Abstract domain lattice for sign analysis

Definition 6 (Logical Representation of Abstract Domains). *A signature* $\Sigma_{\mathcal{A}}$ *for an abstract domain* $\mathcal{A}$ *is a signature where*

- *for every* $a \in \mathcal{A}$ *and every* $z \in \mathbb{Z}$*, there is a constant symbol* $\gamma_{a,z} \in \mathcal{F}$
- *for every* $a \in \mathcal{A}$ *there is a unary predicate symbol* $\chi_a \in \mathcal{P}$

For a signature $\Sigma_{\mathcal{A}}$*, we only consider interpretations* I *satisfying*

- *for every* $a \in \mathcal{A}$ *and every* $z \in \mathbb{Z}$*:* $I(\gamma_{a,z}) \in \gamma(a)$
- *for every* $a \in \mathcal{A}$*:* $I(\chi_a) = \gamma(a)$

The constant symbols $\gamma_{a,z}$ are used to represent abstract values in logical formulas, in particular, on the right hand side of updates. For example, using the sign lattice abstract domain from Ex. 3, the update $(\mathtt{n} := \gamma_{\geq,1} \parallel \mathtt{i} := \gamma_{\geq,2})$ sets `n` and `i` to unknown, not necessarily equal, non-negative values. The predicate symbols

χ_a are used to express membership of a concrete value in the concretization of an abstract value.

For working with the partially interpreted $\gamma_{a,z}$ and χ_a symbols, we need rules for handling them; e.g., we want to be able to prove the validity of a sequent such as $\neg\gamma_{\geq,1} \doteq 0 \Longrightarrow \gamma_{\geq,1} > 0$, which depends on the restriction that $I(\gamma_{\geq,1}) \geq 0$ for every interpretation I. We assume that these rules are provided together with the abstract domain. From now on, we assume a fixed signature $\Sigma_{\mathcal{A}}$ for an abstract domain $\mathcal{A}$.

3.2 Update Weakening and Abstraction Rule

In this section we extend the classical notion of logical weakening to updates for which we give a weakening rule. Update weakening is used in the loop invariant rule directly and also implicitly during loop invariant computation.

To formulate weakening, respectively, strengthening rules for updates, we need to say what *weaker*, respectively, *stronger* means for updates. We define this ordering here with respect to a given sequent proof P and a set of context formulas (or constraints) C.

Definition 7 ($\triangleleft_{P,C}$-relation on updates). *Let P denote a proof, $\mathcal{U}_1$ and $\mathcal{U}_2$ updates, and C a set of formulas. We call $\mathcal{U}_2$ P,C-*weaker *than $\mathcal{U}_1$, i.e.,*

$$\mathcal{U}_1 \triangleleft_{P,C} \mathcal{U}_2$$

if for any interpretation I, state s, and variable assignment β, where for all $\psi \in C$ we have $val_{I,s,\beta}(\psi) = tt$, the following holds:

$$val_{I,s,\beta}(\mathcal{U}_1) \in \{val_{I',s,\beta}(\mathcal{U}_2) \mid I \simeq_{P,C} I'\}$$

where $I \simeq_{P,C} I'$ means that I and I' coincide on all function and predicate symbols occurring in P or C.[3] *In case of an empty set of context formulas C, we omit C and write P-weaker and $\triangleleft_P$ instead.*

Example 4. Assume a proof P consisting of a single sequent

$$c > 0 \Longrightarrow \{\underbrace{\mathtt{i} := \mathtt{i} + 1 \parallel \mathtt{j} := c + 3}_{\mathcal{U}}\}\varphi$$

with program variables $\mathtt{i}, \mathtt{j}$ and a constant symbol c.

1. The update $\mathtt{i} := d + 1 \parallel \mathtt{j} := e$, where d, e are new constant symbols, is P-weaker than $\mathcal{U}$, because for any I, s, β, we can choose the interpretation $I' \simeq_P I$ with $I'(d) = s(\mathtt{i})$ and $I'(e) = I(c) + 3$.
2. The update $\mathtt{i} := f(1) \parallel \mathtt{j} := g(c, 3)$, where f, g are new function symbols, is P-weaker than $\mathcal{U}$, because for any I, s, β we can choose the interpretation $I' \simeq_P I$ with $I'(f)(1) = s(\mathtt{i}) + 1$ and $I'(g)(I(c), 3) = I(c) + 3$.

[3] Note that in particular $val_{I',s,\beta}(\psi) = tt$ holds for all $\psi \in C$.

3. The update $\mathtt{i} := \mathtt{j} \parallel \mathtt{j} := c + 3$ is *not* P-weaker than $\mathcal{U}$, as for any s' with $s'(\mathtt{j}) \neq s(\mathtt{i}) + 1$ the membership requirement from Def. 7 does not hold.
4. The update $\mathtt{i} := \gamma_{\top,0} \parallel \mathtt{j} := \gamma_{pos,0}$, where $\gamma_{\top,0}$ and $\gamma_{pos,0}$ are new, is *not* P-weaker than $\mathcal{U}$, but it is $\{c > 0\}, P$-weaker.

Weakening by replacing the right hand side of an update with a suitable $\gamma_{a,z}$ symbol corresponds to abstracting to the chosen abstract domain. In the following, we restrict ourselves to this form of weakening. The rule weakenUpdate below allows to use it in a sequent proof:

$$\textsf{weakenUpdate}\ \frac{\Gamma, \{\mathcal{U}\}(\bar{\mathtt{x}} \doteq \bar{c}) \Longrightarrow \exists \bar{\gamma}.\{\mathcal{U}'\}(\bar{\mathtt{x}} \doteq \bar{c}), \Delta \qquad \Gamma \Longrightarrow \{\mathcal{U}'\}\varphi, \Delta}{\Gamma \Longrightarrow \{\mathcal{U}\}\varphi, \Delta}$$

where

- $\bar{\mathtt{x}} = (\mathtt{x}_1, \ldots, \mathtt{x}_n)$ is a list of all program variables occurring on the left hand side in $\mathcal{U}$ or $\mathcal{U}'$ (duplicate-free, in an arbitrary order)
- $\bar{c} = (c_1, \ldots, c_n)$ is a list of fresh constant symbols of the same length as $\bar{\mathtt{x}}$
- $\bar{\gamma} = (\gamma_{a_1,z_1}, \ldots, \gamma_{a_m,z_m})$ is a list of all $\gamma_{a,z}$ symbols introduced freshly in $\mathcal{U}'$
- the notation $\exists \bar{\gamma}.\psi$ is an abbreviation for $\exists \bar{y}.(\chi_{\bar{a}}(\bar{y})\ \&\ \psi[\bar{\gamma}/\bar{y}])$, where $\bar{y} = (y_1, \ldots, y_m)$ is a list of fresh logical variables of the same length as $\bar{\gamma}$, and where $\psi[\bar{\gamma}/\bar{y}]$ stands for the formula obtained from ψ by replacing all occurrences of a symbol in $\bar{\gamma}$ with its counterpart in $\bar{y}$
- vector notation is used as an abbreviation: $\exists \bar{y}.\psi$ stands for the multiply quantified formula $\exists y_1.\cdots.\exists y_m.\psi$, $\bar{t} \doteq \bar{t}'$ and $\chi_{\bar{a}}(\bar{y})$ stand for the conjunctions $t_1 \doteq t_1' \ \&\ \cdots \ \&\ t_n \doteq t_n'$ resp. $\chi_{a_1}(y_1) \ \&\ \cdots \ \&\ \chi_{a_m}(y_m)$

The first premiss of weakenUpdate guarantees that $\mathcal{U}'$ is $(P, \Gamma \cup\, !\,\Delta)$-weaker than $\mathcal{U}$: for any initial I, s, β, it must be possible to choose an interpretation of the newly introduced $\bar{\gamma}$ such that with this interpretation, $\mathcal{U}'$ assigns to all relevant program variables $\bar{\mathtt{x}}$ the same value as $\mathcal{U}$. In the second premiss, the proof of φ continues with the weaker update $\mathcal{U}'$ in place of $\mathcal{U}$.

Lemma 1. *The* weakenUpdate *rule is sound: if all of its premisses are logically valid, then its conclusion is also logically valid.*

The proof of this lemma is contained in App. C.1.

3.3 An Invariant Rule Based on Updates

Below we define a variation of the classical loop invariant rule, based on updates. The rule makes use of an "invariant update" $\mathcal{U}'$, which must be provided instead of an invariant formula.

$$\textsf{invariantUpdate}\ \frac{\begin{array}{l}\Gamma, \{\mathcal{U}\}(\bar{\mathtt{x}} \doteq \bar{c}) \Longrightarrow \exists \bar{\gamma}.\{\mathcal{U}'\}(\bar{\mathtt{x}} \doteq \bar{c}), \Delta \\ \Gamma, \{\mathcal{U}'\}g,\ \{\mathcal{U}'\}[\mathtt{p}](\bar{\mathtt{x}} \doteq \bar{c}) \Longrightarrow \exists \bar{\gamma}.\{\mathcal{U}'\}(\bar{\mathtt{x}} \doteq \bar{c}), \Delta \\ \Gamma, \{\mathcal{U}'\}\,!\,g \Longrightarrow \{\mathcal{U}'\}[\ldots]\varphi, \Delta\end{array}}{\Gamma \Longrightarrow \{\mathcal{U}\}[\texttt{while (}g\texttt{) \{p\}; ...}]\varphi, \Delta}$$

where $\bar{\mathtt{x}}$, $\bar{c}$, $\bar{\gamma}$, $\exists\bar{\gamma}.\psi$ and the vector notation are defined as in the weakenUpdate rule.

The first premiss of invariantUpdate is identical to that of weakenUpdate. It ensures that $\mathcal{U}'$ is weaker than $\mathcal{U}$, or in other words, that the initial state for the loop, as produced by $\mathcal{U}$, can also be reached by executing $\mathcal{U}'$ (using some suitable interpretation of the fresh $\bar{\gamma}$ symbols). The second premiss states that $\mathcal{U}'$ is "preserved" by the loop body p: for any state reached by executing first $\mathcal{U}'$ and then p, we can find an interpretation of the $\bar{\gamma}$ such that $\mathcal{U}'$ directly produces this state. Together, the first two premisses establish an inductive argument: any state reachable by an arbitrary number of loop iterations can also be reached directly by $\mathcal{U}'$, for some interpretation of the $\bar{\gamma}$ symbols. The result of this induction is used in the third premiss, where for handling the trail program "..." we only have to consider runs starting in states which can be produced by $\mathcal{U}'$.

Example 5. The following sequent occurs after applying the assignment rule in Ex. 1:

$$\mathtt{i} \geq 0 \Longrightarrow \{\mathtt{n} := 0\}[\mathtt{while\ (i>0)\ } \underbrace{\mathtt{\{i\ =\ i-1;\ n\ =\ n+i\}}}_{\mathtt{b}}](\mathtt{i} \doteq 0\ \&\ \mathtt{n} \geq 0)$$

An appropriate choice for the "invariant update" is $\mathcal{U}' = (\mathtt{n} := \gamma_{\geq,1} \parallel \mathtt{i} := \gamma_{\geq,2})$. We will later see how this update can be found automatically. With this choice, the rule produces the following three sequents:

$$\mathtt{i} \geq 0,\ \{\mathtt{n} := 0\}(\mathtt{n} \doteq c_1\ \&\ \mathtt{i} \doteq c_2) \Longrightarrow \exists y_1, y_2.\big(\chi_{\geq}(y_1)\ \&\ \chi_{\geq}(y_2)\ \&\ \{\mathtt{n} := y_1 \parallel \mathtt{i} := y_2\}(\mathtt{n} \doteq c_1\ \&\ \mathtt{i} \doteq c_2)\big)$$

$$\mathtt{i} \geq 0,\ \{\mathtt{n} := \gamma_{\geq,1} \parallel \mathtt{i} := \gamma_{\geq,2}\}(\mathtt{i} > 0),\ \{\mathtt{n} := \gamma_{\geq,1} \parallel \mathtt{i} := \gamma_{\geq,2}\}[\mathtt{b}](\mathtt{n} \doteq c_1\ \&\ \mathtt{i} \doteq c_2) \Longrightarrow \exists y_1, y_2.\big(\chi_{\geq}(y_1)\ \&\ \chi_{\geq}(y_2)\ \&\ \{\mathtt{n} := y_1 \parallel \mathtt{i} := y_2\}(\mathtt{n} \doteq c_1\ \&\ \mathtt{i} \doteq c_2)\big)$$

$$\mathtt{i} \geq 0,\ \{\mathtt{n} := \gamma_{\geq,1} \parallel \mathtt{i} := \gamma_{\geq,2}\}\,!(\mathtt{i} > 0) \Longrightarrow \{\mathtt{n} := \gamma_{\geq,1} \parallel \mathtt{i} := \gamma_{\geq,2}\}(\mathtt{i} \doteq 0\ \&\ \mathtt{n} \geq 0)$$

All of these sequents are logically valid, and provided that our calculus contains rules covering the semantics of the $\gamma_{\geq,z}$ and $\chi_{\geq}$ symbols, they are proveable. For the first two, one needs to instantiate the existential quantifiers with c_1 and c_2.

Lemma 2. *The* invariantUpdate *rule is sound.*

The proof of this lemma is contained in App. C.2.

3.4 The Proof Search Strategy

In this section we describe the proof search strategy. The proof search strategy implements the fixed point algorithm for handling loops automatically without needing to be provided with loop invariants by a human user.

As our calculus is not proof confluent, defining a *good* search strategy is crucial. In particular, the proof search strategy needs to choose the right degree of

abstraction and to maintain normal form-like properties of updates, terms and formulas (this is important, for example, to actually find a fixed point).

Depending on the proof context (e.g., main or side proof to compute the loop invariant) we will employ different proof search strategies.

Assume we intend to prove that after the execution of a program p the formula φ holds:

$$\Gamma \Longrightarrow [\mathtt{p}]\varphi, \Delta$$

The proof search strategy acts now like a symbolic interpreter on p and executes assignments (applying rule assignment) as well as conditional statements (ifElse). Note that these rules are *precise* in the sense that no information on the possible poststate is lost.

The critical point in a proof P occurs when a loop statement is encountered and we are faced with a situation similar to

$$\Gamma \Longrightarrow \{\mathcal{U}\}[\texttt{while (}g\texttt{) \{b\}; ...}]\varphi, \Delta$$

In abstract-interpretation approaches, loop treatment involves the computation of a safe approximation of the set of states observable after the loop termination. It remains then to show that the formula $[\ldots]\varphi$ holds in all of them. The main idea of our approach is to describe this set in terms of an (abstract) update $\mathcal{U}_a$ such that for each I, β the set $\{s' \mid s' = val_{I',s,\beta}(\mathcal{U}_a)(s), s \in \mathcal{S}, f.a.\ I' \simeq_{P,C} I\}$ is a safe approximation of the post loop states. A higher precision can be achieved by requiring that the considered interpretations I satisfy additional formulas.

To compute abstract (weaker) updates, the proof search strategy spawns side proofs. The purpose of these side proofs is to compute updates that capture the state changes in successive executions of the loop body. The results of the side proofs are later combined after suitable abstraction using the *join* rule. Consequently, in the side proofs we handle the top-level loop by unwinding (loopUnwind) while possible nested loops are treated by rule invariantUpdate. As in the side proofs we are only interested in state changes, all proof branches that do not involve symbolic execution are discarded.

Consider now one unwinding step: the proof search strategy executes the loop body until the loop is about to be re-entered. In general, symbolic execution of the loop body may result in several branches; the proof search continues on these branches until they are either closed or the loop body has been completely symbolically executed and the loop is about to be re-entered. After complete execution of one loop iteration, the proof situation is similar to the one shown in Fig. 2.

At this point the proof search strategy computes a weakened update representing a superset of all possible states reachable after this loop iteration. A new sequent of the form

$$\Gamma' \Longrightarrow \{\mathcal{U}'\}[\texttt{while (}g\texttt{) \{b\}; ...}]\varphi,\ \Delta'$$

is created where update $\mathcal{U}'$ is the weakened (by abstraction) update computed by comparing the updates $\mathcal{U}_1, \ldots, \mathcal{U}_m$ from the open branches *and* update $\mathcal{U}$

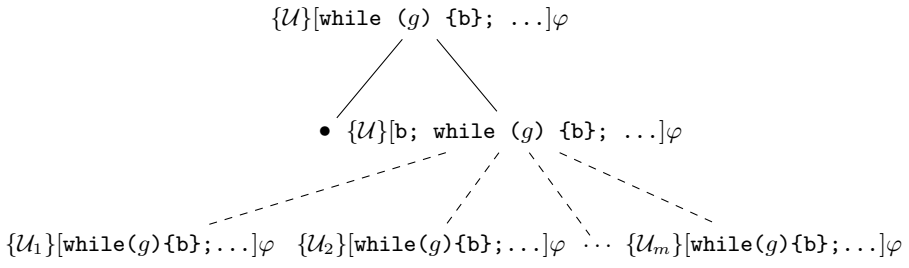


Fig. 2. Invariant computation: side proof after symbolic execution of one loop iteration

representing the symbolic state just before the loop unwinding. Γ' (Δ') are formula sets that are weaker (stronger) than any of the corresponding $\Gamma_1, \ldots, \Gamma_m$ ($\Delta_1, \ldots, \Delta_m$) belonging to the open leaves considered in Fig. 2. In Sect. 3.5 we describe this *join* in detail. The constructed sequent is then appended at one of the open branches. The other branches are closed, i.e. not further taken into consideration.

The proof search strategy stops the side computation if after an application of the join rule a fixed point is detected. A fixed point is reached when update $\mathcal{U}$ taken from immediately before the last loopUnwind rule application is weaker than (or equal to) update $\mathcal{U}'$ resulting from the current join operation.

To detect a fixed point the proof search strategy tries to prove for all program variables $\bar{\mathtt{x}}$ that the states represented by update $\mathcal{U}$ subsume those of $\mathcal{U}'$:

$$\forall \bar{y'}.\exists \bar{y}.(Eq(\bar{y'}, \bar{y}) \;\&\; \chi_{\bar{\gamma'}}(\bar{y'}) \;\texttt{->}\; \chi_{\bar{\gamma}}(\bar{y}) \;\&\; \{\mathcal{U}[\bar{\gamma}/\bar{y}]\}\bar{\mathtt{x}} \doteq \{\mathcal{U}'[\bar{\gamma'}/\bar{y'}]\}\bar{\mathtt{x}}) \qquad (1)$$

where

- $\bar{\gamma}, \bar{\gamma'}$ denote sequences of all γ symbols occurring in one of the sequents
- $\bar{y}, \bar{y'}$ are duplicate-free sequences of variables of same length as $\bar{\gamma}$ resp. $\bar{\gamma'}$
- $Eq(\bar{y'}, \bar{y}) := \bigwedge_{\substack{y_i \in \bar{y}, y'_j \in \bar{y'}, \\ \gamma_{a_i,i} = \gamma_{a_j,j}}} y_i = y'_j$ and $\chi_{\bar{\gamma}}(\bar{y}) := \bigwedge_{\substack{\gamma_{a_i,i} \in \bar{\gamma}, \\ y_i \in \bar{y}}} \chi_{a_i}(y_i)$ (analog. $\chi_{\bar{\gamma'}}$)

To find fixed points earlier the sequent side-formulas Γ, Δ, Γ', Δ' can be used in the proof. The join operation defined in the next section guarantees that if the value of a variable x has been changed in the most recent loop iteration then the abstraction produces an elementary update $\mathtt{x} := \gamma_{k,a}$. In combination with a finite abstract domain (1) becomes trivial to prove such that a fixed point is guaranteed to be found.

3.5 Joining Proof Branches

In this section we describe how different execution paths are joined by the proof search strategy in a side proof. The join rule introduced in this section is a combination of a classical weakening and the update weakening rule. Deviating

from other rules, it is not a sequent rule but a "meta rule" combining several sequents. Let P denote a proof with several open branches

$$\begin{array}{c} \vdots \\ \Gamma_{s_0} \Longrightarrow \{\mathcal{U}_{s_0}\}[\texttt{while (}g\texttt{)\{b\}}]\varphi,\ \Delta_{s_0} \\ \vdots \end{array}$$

$$\Gamma_{s_1} \Longrightarrow \{\mathcal{U}_{s_1}\}[\texttt{while (}g\texttt{)\{b\}}]\varphi,\ \Delta_{s_1}\ \ldots\ \Gamma_{s_m} \Longrightarrow \{\mathcal{U}_{s_m}\}[\texttt{while (}g\texttt{)\{b\}}]\varphi,\ \Delta_{s_m}$$

Applying the join rule closes all except one of these open branches. The open branch that is left is extended by adding the sequent

$$\bigvee_{i=s_0}^{s_m} (\Gamma_{s_i}\ \&\ !\,\Delta_{s_i}) \Longrightarrow \{(C_{s_0},\mathcal{U}_{s_0})\dot{\sqcup}\ldots\dot{\sqcup}(C_{s_m},\mathcal{U}_{s_m})\}[\texttt{while (}g\texttt{) \{b\}}]\varphi$$

as a new leaf with

- formula set $C_{s_i} := \Gamma_{s_i} \cup\, !\,\Delta_{s_i}$ and
- $(C_1,\mathcal{U}_1)\,\dot{\sqcup}\,(C_2,\mathcal{U}_2)$ is an update join operation as defined below.

Definition 8 (Update Join $\cdot\,\dot{\sqcup}\,\cdot$). *The* update join operation *has the signature*

$$\dot{\sqcup} : (2^{For} \times Updates) \times (2^{For} \times Updates) \to Updates$$

where 2^{For} denotes the power set of formulas and is defined by the following property:

Let $\mathcal{U}_1$ and $\mathcal{U}_2$ denote arbitrary updates occurring in a proof P and let C_1, C_2 be formula sets representing constraints on the update values. Then an update $(C_1,\mathcal{U}_1)\,\dot{\sqcup}\,(C_2,\mathcal{U}_2)$ must be $(P, C_{1/2})$-weaker than $\mathcal{U}_1$ resp. $\mathcal{U}_2$, i.e.

$$\mathcal{U}_i \lhd_{P,C_i} (\mathcal{U}_1, C_1)\,\dot{\sqcup}\,(\mathcal{U}_2, C_2),\ i = 1, 2\ .$$

Lemma 3. *Rule* join *is sound.*

The join rule, even though sound, is only used within side proofs that compute loop invariants. Its correctness is not strictly necessary as the loop invariant rule checks the invariance property and will reject unsuitable invariants, but increases the likelihood that meaningful fixed points and, hence, loop invariants are found.

Finally, we describe the concrete realization $\sqcup_{abs}$ of an update join operator for finite abstract domains.

Let $\mathcal{U}_1, C_1$ and $\mathcal{U}_2, C_2$ denote updates and their value restrictions. The update join $(\mathcal{U}_1, C_1)\dot{\sqcup}_{abs}(\mathcal{U}_2, C_2)$ computes the update $\mathcal{U}_{res}$ as follows: let $\mathtt{x}$ be a program variable occurring on the left side of $\mathcal{U}_1$ or $\mathcal{U}_2$.

1. Try to prove

$$\Longrightarrow \exists y.((C_1 \texttt{ -> } (\{\mathcal{U}_1\}\mathtt{x}) \doteq y)\ \&\ (C_2 \texttt{ -> } (\{\mathcal{U}_2\}\mathtt{x}) \doteq y))$$

   if the proof attempt succeeds, then the elementary update $\mathtt{x} := t_1$ occurring last in $\mathcal{U}_1$ with $\mathtt{x}$ on the left side (resp. $\mathtt{x} := t_2$ if $\mathtt{x}$ occurred only on the left side of $\mathcal{U}_2$) is added to $\mathcal{U}_{res}$ by parallel composition. Otherwise, if the proof attempt fails (timeout or counterexample found) then continue with the next step.

2. For each pair $(C_i, \mathcal{U}_i)$, $i = 1, 2$, for any abstract domain element a starting with the smallest one, try to prove

$$C_i \Longrightarrow \chi_a(\{\mathcal{U}_i\}\mathtt{x})$$

and stop processing a pair as soon as an a has been found for which the sequent is valid, i.e. a proof has been found (within a given timeout). After termination we are left with two abstract domain elements a_1, a_2 for the resp. pairs for which we compute $a_1 \sqcup a_2$ (or at least an upper bound). Finally, the elementary update $\mathtt{x} := \gamma_{a_1 \sqcup a_2, z}$ is added to update $\mathcal{U}_{res}$ by parallel composition.

Example 6. Given the program variables $\mathtt{i}, \mathtt{n}$ and the update/constraint pairs $(\mathtt{n} := 0, \mathtt{i} \geq 0)$ and $(\mathtt{n} := \mathtt{i} - 1 \| \mathtt{i} := \mathtt{i} - 1, \mathtt{i} > 0)$, the join computation proceeds as follows:

Starting with program variable $\mathtt{n}$, we check first, if $\mathtt{n}$ is evaluated to the same value under both updates in their resp. context. Obviously, that does not hold in a state where $\mathtt{i}$ has, for example, the value 10.

Thus we enter the abstraction phase. Starting with the minimal abstract domain element $\bot$ the proof obligations described in step 2 are attempted to prove. The attempts succeed for

$$\mathtt{i} \geq 0 \Longrightarrow \chi_0(\{\mathtt{n} := 0\}\mathtt{n}) \quad \text{and} \quad \mathtt{i} > 0 \Longrightarrow \chi_{\geq}(\{\mathtt{n} := \mathtt{i} - 1 \parallel \mathtt{i} := \mathtt{i} - 1\}\mathtt{n})$$

The join for the abstract domain elements is $(\geq \sqcup\, 0) = \geq$. Thus, we get as first sub-update $\mathtt{n} := \gamma_{\geq,0}$. A similar computation for program variable $\mathtt{i}$ gives us finally the complete update

$$\mathtt{n} := \gamma_{\geq,0} \parallel \mathtt{i} := \gamma_{\geq,1}$$

4 Example

Recall the proof obligation from Ex. 1:

$$\mathtt{i} \geq 0 \Longrightarrow [\mathtt{n = 0; \ while \ (i > 0) \ i = i\text{-}1; \ n = n+i}](\mathtt{i} \doteq 0 \ \& \ \mathtt{n} \geq 0) \quad (2)$$

In this section, we illustrate our approach by slowly walking through the proof for this sequent. We abbreviate the while-loop with `W`, the loop body with `B` and the postcondition with φ. The first step is to apply the assignment rule, which produces the following sequent:

$$\mathtt{i} \geq 0 \Longrightarrow \{\mathtt{n} := 0\}[\mathtt{W}]\varphi \quad (3)$$

At this point we open a side computation with this subgoal in order to determine a suitable loop invariant update. After this side computation, we will return to the main proof at sequent (3) and apply the invariantUpdate rule using this update.

The side computation starts by applying loopUnwind, which splits the side proof into two branches:

$$\mathtt{i} \geq 0,\ \{\mathtt{n} := 0\}(\mathtt{i} > 0) \Longrightarrow \{\mathtt{n} := 0\}[\mathtt{B;W}]\varphi$$

$$\mathtt{i} \geq 0,\ \{\mathtt{n} := 0\}\,!(\mathtt{i} > 0) \Longrightarrow \{\mathtt{n} := 0\}\varphi$$

The second of these branches is uninteresting to us in this side computation, and we simply ignore it. Using update rewriting rules and some arithmetic simplification, the first branch can be simplified to

$$\mathtt{i} > 0 \Longrightarrow \{\mathtt{n} := 0\}[\mathtt{B;W}]\varphi$$

Note that the path condition from the loop guard strengthens the precondition. We continue by symbolically executing the loop body, which (after some update rewriting) yields

$$\mathtt{i} > 0 \Longrightarrow \{\mathtt{n} := 0 \parallel \mathtt{i} := \mathtt{i} - 1 \parallel \mathtt{n} := 0 + \mathtt{i} - 1\}[\mathtt{W}]\varphi \tag{4}$$

Now, we have completed our first iteration: we have unwound the loop once, executed its body, and obtained a sequent where W is the first active statement like in (3). We use the join rule to merge the current state (4) with the previous state (3):

$$\mathtt{i} > 0 \mid \mathtt{i} \geq 0 \Longrightarrow \{\mathtt{n} := \gamma_{\geq,1} \parallel \mathtt{i} := \gamma_{\geq,2}\}[\mathtt{W}]\varphi \tag{5}$$

The computation performed by join rule is explained in detail in Ex. 6 in Sect. 3.5. We unwind the loop once more with loopUnwind, which gives us the following for the loop entry branch:

$$\mathtt{i} > 0 \mid \mathtt{i} \geq 0, \{\mathtt{n} := \gamma_{\geq,1} \parallel \mathtt{i} := \gamma_{\geq,2}\}(\mathtt{i} > 0) \Longrightarrow \{\mathtt{n} := \gamma_{\geq,1} \parallel \mathtt{i} := \gamma_{\geq,2}\}[\mathtt{B;W}]\varphi$$

Update rewriting and arithmetic simplification turns this into:

$$\mathtt{i} \geq 0,\ \gamma_{\geq,2} > 0 \Longrightarrow \{\mathtt{n} := \gamma_{\geq,1} \parallel \mathtt{i} := \gamma_{\geq,2}\}[\mathtt{B;W}]\varphi$$

We symbolically execute the body a second time, which produces:

$$\begin{aligned} &\mathtt{i} \geq 0,\ \gamma_{\geq,2} > 0 \\ &\Longrightarrow \{\mathtt{n} := \gamma_{\geq,1} \parallel \mathtt{i} := \gamma_{\geq,2} \parallel \mathtt{i} := \gamma_{\geq,2} - 1 \parallel \mathtt{n} := \gamma_{\geq,1} + \gamma_{\geq,2} - 1\}[\mathtt{W}]\varphi \end{aligned} \tag{6}$$

This finishes our second iteration. We apply join to combine (6) and (5), which yields:

$$\mathtt{i} > 0 \mid \mathtt{i} \geq 0 \mid (\mathtt{i} \geq 0 \ \&\ \gamma_{\geq,2} > 0) \Longrightarrow \{\mathtt{n} := \gamma_{\geq,3} \parallel \mathtt{i} := \gamma_{\geq,4}\}[\mathtt{W}]\varphi \tag{7}$$

Now, we observe that the update $\mathcal{U} = (\mathtt{n} := \gamma_{\geq,3} \parallel \mathtt{i} := \gamma_{\geq,4})$ of the current sequent (7) "implies" the corresponding update $\mathcal{U}' = (\mathtt{n} := \gamma_{\geq,1} \parallel \mathtt{i} := \gamma_{\geq,2})$ of the previous iteration (5). The fixed point detection formula (1) from Sect. 3.4

$$\begin{aligned} &\forall y_1, y_2.\exists y_1', y_2'.\big(\\ &\quad \chi_{\geq}(y_1)\ \&\ \chi_{\geq}(y_2) \texttt{->} (\chi_{\geq}(y_1')\ \&\ \chi_{\geq}(y_2')\ \& \\ &\qquad ((\{\mathtt{n} := y_1' \parallel \mathtt{i} := y_2'\}n) \doteq (\{\mathtt{n} := y_1 \parallel \mathtt{i} := y_2\}n)\ \& \\ &\qquad\ (\{\mathtt{n} := y_1' \parallel \mathtt{i} := y_2'\}i) \doteq (\{\mathtt{n} := y_1' \parallel \mathtt{i} := y_2'\}i))\big) \end{aligned}$$

becomes then trivial to solve as the existential quantifiers need only to be instantiated with the skolem constant resulting from the enclosing universal quantifier.

Thus, $\mathcal{U}$ (or $\mathcal{U}'$) is a "fixed point". At this point we leave the side computation. We continue the main proof by applying the rule invariantUpdate to (3), which eliminates the loop from our proof obligation, making the remainder of the proof straightforward as shown already in Ex. 5.

In conclusion, we have constructed a proof for the validity of (2). Our use of abstraction allowed us to do so in a completely mechanical process, which did not require any manually supplied loop invariant.

5 Modeling Information Flow

The problem of *information flow security* is about preventing a program from leaking "secret" data to output channels of a "lower security level". Typically, the security levels to be distinguished are defined and ordered in a security lattice. In the simplest case, one distinguishes only between the security levels High and Low.

Example 7. In the following example programs, `h` and `l` are program variables, where `h` has security level High and `l` security level Low. A program is considered *secure* if an attacker who reads the final values of the Low variables cannot infer any information about the initial values of the High variables.

1. `l=h` is obviously *insecure*, because information flows directly from `h` to `l`.
2. `if (h>0) {l=1} else {l=2}` is also *insecure*, because information about the sign of the initial value of `h` flows indirectly to `l`.
3. `if (l>0) {h=1} else {h=2}` is *secure*, because the value of `l` is not touched at all.
4. `if (h>0) {l=1} else {l=2}; l=3` is *secure*, because the final value of `l` is always 3, independently of the initial value of `h`.
5. `h=0;l=h` is *secure*, because the final value of `l` is always 0.
6. `if (h>0) {h=l;l=h}` is *secure*, because the value of `l` is not changed.
7. `if (h>0) {l=2;h=1} else {l=2;h=2}` is *secure*, because the final value of `l` is always 2.
8. `l=h-h` is *secure*, because the final value of `l` is always 0.

The most common technique for a language-based analysis of information flow is to use special type systems. The security levels are then used as types that are assigned to program variables. The analysis ensures via type checking or type inference that no information about the value of a High-labeled variable is leaked to a Low-labeled variable.

Soundness of any approach to information-flow analysis entails that an insecure program will not be classified as secure. To achieve full automation, however, many approaches, in particular type-based ones, classify certain secure programs as insecure. To identify program (4) as secure, the approach under consideration has to be control-flow sensitive. Some, but not all, available analyses have this property. In order to correctly identify programs (5), (6), (7) and (8) as secure,

the analysis must be *value-sensitive*. At the moment this is only achieved by some deduction-based systems [9,6,13] that require human interaction.

Information-flow analysis can be restated as an analysis of variable dependencies (see [16]). Here, we want to find for any variable $\mathtt{x}$ the set of variables on whose initial values the final value of $\mathtt{x}$ can at most depend. In particular, we may ask whether the final value of a Low-labeled variable can depend on the initial value of any High-labeled variable.

In this section we extend our program logic to allow the analysis of variable dependencies in programs. In contrast to [9], where the dependencies of a program variable are implicitly tracked using free logical variables, we use an approach where the dependencies are encoded explicitly into program states. The execution of a program statement directly manipulates these dependencies. This approach allows to apply the abstraction mechanism introduced in this paper also to variable dependencies, which serves to achieve *automation* of our information flow analysis while maintaining a high degree of precision and achieving value-sensitivity in more cases than type-based systems.

We omitted formal correctness statements and proofs in this section which are tedious, but do not offer additional insights.

5.1 Dependencies in Dynamic Logic

Formally, the dependencies of a variable can be defined as follows.

Definition 9 (Variable Dependencies). *Given a program variable* $\mathtt{x}$ *and a program* $\mathtt{p}$*, the* dependencies of $\mathtt{x}$ under $\mathtt{p}$ *form the smallest set* $\mathcal{D}(\mathtt{x},\mathtt{p}) \subseteq \mathcal{PV}$ *of program variables such that the following holds for all interpretations* I *and all variable assignments* β*: if* $s_1, s_2 \in \mathcal{S}$ *are such that for all* $\mathtt{y} \in \mathcal{D}(\mathtt{x},\mathtt{p})$ *we have* $s_1(\mathtt{y}) = s_2(\mathtt{y})$*, then either*

- $val_{I,s_1,\beta}(\mathtt{p}) = val_{I,s_2,\beta}(\mathtt{p}) = \emptyset$ *(i.e., from both initial states the execution of* $\mathtt{p}$ *does not terminate), or*
- $val_{I,s_1,\beta}(\mathtt{p}) = \{s_1'\}$ *and* $val_{I,s_2,\beta}(\mathtt{p}) = \{s_2'\}$*, where* $s_1'(\mathtt{x}) = s_2'(\mathtt{x})$ *(i.e., from both initial states the execution terminates and yields the same value for* $\mathtt{x}$*).*

The dependencies formalized in Def. 9 are difficult to reason about: they are based on comparing *all possible runs* of a program $\mathtt{p}$ instead of being a local property which is true or false in a given program state. To be able to talk about dependencies in our logical formulas in the same way as about other program properties, we extend our logic and the semantics of programs so that dependencies are stored in states *explicitly*. The main idea is to associate with every program variable $\mathtt{x}$ a program variable $\mathtt{x}^{dep}$ that records the dependencies of $\mathtt{x}$ with respect to the program that has been symbolically executed so far. The variable $\mathtt{x}^{dep}$ is updated by the program whenever $\mathtt{x}$ itself is changed, such that in any state during program execution, $\mathtt{x}^{dep}$ evaluates to a set of program variables which contains all variables on whose initial value the current value of $\mathtt{x}$ can depend.

Definition 10 (Logical Representation of Dependencies). *Given a signature $\Sigma = (\mathcal{F}, \mathcal{P}, \mathcal{PV}, \mathcal{V})$, the* dependency extension *of Σ is a signature $\Sigma^{dep} = (\mathcal{F}^{dep}, \mathcal{P}^{dep}, \mathcal{PV}^{dep}, \mathcal{V})$, where*

- $\mathcal{F}^{dep} = \mathcal{F} \cup \{\texttt{\{\}}, \dot\cup\} \cup \{\texttt{\{x\}} \mid \texttt{x} \in \mathcal{PV}\}$, *where* `{}` *is a constant symbol,* $\dot\cup$ *is a function symbol with arity 2, and where the* `{x}` *are function symbols with arity 0,*
- $\mathcal{P}^{dep} = \mathcal{P} \cup \{\dot\subseteq\}$, *where* $\dot\subseteq$ *is a predicate symbol with arity 2, and*
- $\mathcal{PV}^{dep} = \mathcal{PV} \cup \{\texttt{x}^{dep} \mid \texttt{x} \in \mathcal{PV}\}$.

For such a signature Σ^{dep}, we do not allow the new symbols to occur in programs: programs over a signature Σ^{dep} are built only from the symbols defined in the sub-signature Σ. We only consider universes $\mathsf{D} \supseteq 2^{\mathcal{PV}}$ where every set of program variables also occurs as a value in the universe. Finally, we only allow interpretations I that fix the meaning of the additional symbols as follows:

- $I(\texttt{\{\}}) = \emptyset$,
- *for all* $P_1, P_2 \in 2^{\mathcal{PV}}$: $I(\dot\cup)(P_1, P_2) = P_1 \cup P_2$,
- *for all* $\texttt{x} \in \mathcal{PV}$: $I(\texttt{\{x\}}) = \{\texttt{x}\}$, *and*
- $I(\dot\subseteq) = \{(P_1, P_2) \mid P_1 \subseteq P_2 \subseteq \mathcal{PV}\}$.

Definition 11 (Program Semantics with Dependencies). *Given a universe* D, *an interpretation I, a state s and a variable assignment β, we evaluate programs* `p` *to a set of states $val'_{I,s,\beta}(\texttt{p}) \in 2^{\mathcal{S}}$ as defined in App. A.2. As before, our programs are deterministic, so the sets always have at most one element.*

One difference to the program semantics without dependencies is that executing an assignment `x = t` not only changes `x`, but also $\texttt{x}^{dep}$: we assign to it the value of $deps(t)$, where for every term or formula t, $deps(t)$ is a term which over-approximates the precise semantic dependencies of t. For example, $deps(\texttt{n+i}) = \texttt{n}^{dep} \dot\cup \texttt{i}^{dep}$. The formal definition of *deps* is given in App. A.3.

The second difference is that after executing a conditional statement or a loop iteration with guard g, we add $deps(g)$ to $\texttt{x}^{dep}$ for every program variable `x` which has been changed in the body of the conditional or loop. This is necessary in order to cover implicit flow of information via control flow (see Ex. 7).

Example 8. Consider program (6) of Ex. 7. We can express security of this program with the sequent

$$\texttt{h}^{dep} \doteq \texttt{\{h\}},\ \texttt{l}^{dep} \doteq \texttt{\{l\}} \Longrightarrow [\texttt{if (h>0) \{h=l;l=h\}}](\texttt{l}^{dep} \dot\subseteq \texttt{\{l\}})$$

The precondition in the antecedent means that we assume the initial value of every variable to depend exactly on itself. The postcondition demands that after running the program, the final value of `l` depends at most on the initial value of `l` (so that in particular, it does not depend on the initial value of `h`).

Let $s_1 \in \mathcal{S}$ be a state satisfying the precondition, i.e., $s_1(\texttt{h}^{dep}) = \{\texttt{h}\}$ and $s_1(\texttt{l}^{dep}) = \{\texttt{l}\}$. If we execute the assignment `h=l` in s_1, this will produce a state s_2 with $s_2(\texttt{h}^{dep}) = s_1(\texttt{l}^{dep}) = \{\texttt{l}\}$, reflecting the fact that now the value of `h` depends on the initial value of `l`.

Continuing the execution of the program, the assignment `l=h` yields a state s_3 with $s_3(\mathtt{l}^{dep}) = s_2(\mathtt{h}^{dep}) = \{\mathtt{l}\}$. After the end of the conditional statement, the dependencies of the guard `h>0` are injected into all variables changed inside the conditional, yielding a state s_4 where $s_4(\mathtt{l}^{dep}) = s_3(\mathtt{l}^{dep}) = \{\mathtt{l}\}$ (since `l` has the same value in s_3 at the end of the conditional as it had in s_1 before the conditional), and where $s_4(\mathtt{h}^{dep}) = s_3(\mathtt{h}^{dep}) \cup s_1(\mathtt{h}^{dep}) = \{\mathtt{h}\}$ (where $s_1(\mathtt{h}^{dep})$ are the dependencies of the guard).

Thus, the final state s_4 satisfies $s_4(\mathtt{l}^{dep}) = \{\mathtt{l}\}$, meaning that the postcondition is satisfied. As this holds for all initial states satisfying the precondition, our sequent is logically valid.

Note that our formalisation of dependencies is control flow- and value-sensitive; it correctly classifies programs (3)–(6) of Ex. 7 as secure. Nevertheless, it is an overapproximation of the semantic dependencies as formalized in Def. 9. For example, it conservatively classifies programs (7) and (8) as insecure, even though they are in fact secure. This is a price we pay for the ability to reason about dependencies in the same way as state properties.

5.2 Dependency Aware Rules

For working with the changed semantics of Def. 11 in our calculus, we need to adapt the symbolic execution rules from Sect. 2.3 and also the update invariant rule from Sect. 3.3 accordingly. The other rules (in particular, weakenUpdate and join) are not affected, because they do not deal with programs. For the assignment rule, we can simply add the update $\mathtt{x}^{dep} := deps(t)$:

$$\mathsf{assignment}^{\mathrm{dep}}\ \frac{\Gamma \Longrightarrow \{\mathcal{U}\}\{\mathtt{x} := t \parallel \mathtt{x}^{dep} := deps(t)\}[\ldots]\varphi, \Delta}{\Gamma \Longrightarrow \{\mathcal{U}\}[\mathtt{x = t;\ \ldots}]\varphi, \Delta}$$

For conditional statements, the new semantics introduces an additional state transition after execution of the conditional where the dependencies of the guard are retroactively added to the dependencies of all variables modified inside the conditional. We capture these additional dependencies in the rule by inserting a suitable update $\mathcal{V}$ into our premises:

$$\mathsf{ifElse}^{\mathrm{dep}}\ \frac{\begin{array}{c}\Gamma, \{\mathcal{U}\}g,\ \{\mathcal{U}\}(\bar{\mathtt{y}} \doteq \bar{y}^{pre}) \Longrightarrow \{\mathcal{U}\}[\mathtt{p1}]\{\mathcal{V}\}[\ldots]\varphi, \Delta \\ \Gamma, \{\mathcal{U}\}\,!\,g,\ \{\mathcal{U}\}(\bar{\mathtt{y}} \doteq \bar{y}^{pre}) \Longrightarrow \{\mathcal{U}\}[\mathtt{p2}]\{\mathcal{V}\}[\ldots]\varphi, \Delta\end{array}}{\Gamma \Longrightarrow \{\mathcal{U}\}[\mathtt{if\ (}g\mathtt{)\ \{p1\}\ else\ \{p2\};\ \ldots}]\varphi, \Delta}$$

where

- $\bar{\mathtt{y}} = (\mathtt{y}_1, \mathtt{y}_1^{dep}, \ldots, \mathtt{y}_n, \mathtt{y}_n^{dep})$ is a list of all program variables occurring in g, `p1` or `p2`, together with the corresponding dependency variables
- $\bar{y}^{pre} = (y_1^{pre}, y_1^{predep} \ldots, y_n^{pre}, y_n^{predep})$ is a list of fresh constant symbols of the same length as $\bar{\mathtt{y}}$
- $\mathcal{V}$ is the update

$$\mathtt{y}_1^{dep} := \mathit{if}(\mathtt{y}_1 \doteq y_1^{pre})\mathit{then}(\mathtt{y}_1^{dep})\mathit{else}\big(\mathtt{y}_1^{dep} \dot\cup \{\bar{\mathtt{y}} := \bar{y}^{pre}\}\mathit{deps}(g)\big)$$
$$\| \ldots \|$$
$$\mathtt{y}_n^{dep} := \mathit{if}(\mathtt{y}_n \doteq y_n^{pre})\mathit{then}(\mathtt{y}_n^{dep})\mathit{else}\big(\mathtt{y}_n^{dep} \dot\cup \{\bar{\mathtt{y}} := \bar{y}^{pre}\}\mathit{deps}(g)\big)$$

The fresh constant symbols $\bar{y}^{pre}$ are used to store the pre-state values of the program variables $\bar{\mathtt{y}}$. The update $\mathcal{V}$ compares the current values of all (non-dependency) program variables with their pre-state values, and adds $\mathit{deps}(g)$ to variable's dependencies if the value has been changed. A subtle detail is that $\mathit{deps}(g)$ must be evaluated in the pre-state, which is achieved by prefixing it with the update $\bar{\mathtt{y}} := \bar{y}^{pre}$.

The same idea can be applied to the loopUnwind and invariantUpdate rules (where $\bar{\mathtt{y}}$, $\bar{y}^{pre}$ and $\mathcal{V}$ are as above):

$$\textsf{loopUnwind}^{\text{dep}}\quad \frac{\Gamma, \{\mathcal{U}\}g,\ \{\mathcal{U}\}(\bar{\mathtt{y}} \doteq \bar{y}^{pre}) \Longrightarrow \{\mathcal{U}\}[\mathtt{p}]\{\mathcal{V}\}[\texttt{while (}t\texttt{) \{p\}; ...}]\varphi, \Delta \quad \Gamma, \{\mathcal{U}\}\,!\,g \Longrightarrow \{\mathcal{U}\}[\ldots]\varphi, \Delta}{\Gamma \Longrightarrow \{\mathcal{U}\}[\texttt{while (}g\texttt{) \{p\}; ...}]\varphi, \Delta}$$

$$\textsf{invariantUpdate}^{\text{dep}}\quad \frac{\Gamma, \{\mathcal{U}\}(\bar{\mathtt{x}} \doteq \bar{c}) \Longrightarrow \exists\bar{\gamma}.\{\mathcal{U}'\}(\bar{\mathtt{x}} \doteq \bar{c}), \Delta \quad \Gamma, \{\mathcal{U}'\}g,\ \{\mathcal{U}'\}(\bar{\mathtt{y}} \doteq \bar{y}^{pre}),\ \{\mathcal{U}'\}[\mathtt{p}]\{\mathcal{V}\}(\bar{\mathtt{x}} \doteq \bar{c}) \Longrightarrow \exists\bar{\gamma}.\{\mathcal{U}'\}(\bar{\mathtt{x}} \doteq \bar{c}), \Delta \quad \Gamma, \{\mathcal{U}'\}\,!\,g \Longrightarrow \{\mathcal{U}'\}[\ldots]\varphi, \Delta}{\Gamma \Longrightarrow \{\mathcal{U}\}[\texttt{while (}g\texttt{) \{p\}; ...}]\varphi, \Delta}$$

5.3 Dependency Aware Abstraction

To apply abstraction in the dependency-aware version of our calculus two abstract domains have to be defined:

1. The abstract domain $\mathcal{A}_{val}$ for the value abstraction of *normal* program variables that carry values. The choice of $\mathcal{A}_{val}$ depends on the application context. An example is the sign domain for integers used for illustration in the previous sections.
2. The abstract domain $\mathcal{A}_{dep}$ for the value abstraction of the *dependency* program variables. Again, the suitable choice depends on the application context. In an information-flow security context, a natural choice for $\mathcal{A}_{dep}$ is suggested by the security lattice.

The proof search strategy remains nearly unchanged from the standard version as defined in Sec. 3.4. When computing an abstraction, the abstract domain $\mathcal{A}_{val}$ is used for normal program variables $\mathtt{x} \in \mathcal{PV}$ and $\mathcal{A}_{dep}$ for dependency program variables $\mathtt{x}^{dep} \in \mathcal{PV}^{dep}$.

Example 9. Assume that we are given a security policy with security levels High and Low for program variables $\mathcal{PV} = \{\mathtt{l1}, \mathtt{l2}, \mathtt{h}\}$.

For regular values, we keep using the sign domain from previous sections as $\mathcal{A}_{val}$. For dependencies, we use $\mathcal{A}_{dep} := \{\emptyset, \mathsf{Low}, \mathsf{High}, \top_{dep}\}$ with $\gamma(\emptyset) = \emptyset$, $\gamma(\mathsf{Low}) = 2^{\{\mathtt{l1}, \mathtt{l2}\}}$, $\gamma(\mathsf{High}) = 2^{\{\mathtt{h}\}}$, $\gamma(\top_{dep}) = 2^{\mathcal{PV}}$. Consider now the following simple program `P`:

```
l1=0; l2=0; while (h<0) { l2=l2+1; h=h+1 }; if (l2<0) { l1=1 }
                          W                    C
```

To check whether `P` satisfies the specified security policy for program variable `l1`, the sequent

$$\mathtt{l1}^{dep} \doteq \{\mathtt{l1}\},\ \mathtt{l2}^{dep} \doteq \{\mathtt{l2}\},\ \mathtt{h}^{dep} \doteq \{\mathtt{h}\} \Longrightarrow [\mathtt{P}]\big(\mathtt{l1}^{dep} \dot{\subseteq} \{\mathtt{l1}\} \dot{\cup} \{\mathtt{l2}\}\big)$$

needs to be proven. The precondition demands that program variables depend on themselves in the initial state, as in Ex. 8.

Applying rule $\mathsf{assignment}^{\mathrm{dep}}$ twice yields an update where `l1` and `l2` are set to 0 and where all dependencies of `l1` and `l2` have been erased, i.e. $\mathtt{l1}^{dep}$, $\mathtt{l2}^{dep}$ are assigned the empty set `{}` ($= deps(0)$). The resulting sequent is:

$$\mathtt{l1}^{dep} \doteq \{\mathtt{l1}\},\ \mathtt{l2}^{dep} \doteq \{\mathtt{l2}\},\ \mathtt{h}^{dep} \doteq \{\mathtt{h}\}$$
$$\Longrightarrow \{\mathtt{l1} := 0 \parallel \mathtt{l1}^{dep} := \{\} \parallel \mathtt{l2} := 0 \parallel \mathtt{l2}^{dep} := \{\}\}[\mathtt{W;C}]\big(\mathtt{l1}^{dep} \dot{\subseteq} \{\mathtt{l1}\} \dot{\cup} \{\mathtt{l2}\}\big)$$

At this point the loop invariant update needs to be computed in a side proof as described in Sect. 3.4, which automatically yields as invariant update $\mathcal{U}_{Inv}$:

$$\mathtt{l1} := 0 \parallel \mathtt{l1}^{dep} := \{\} \parallel \mathtt{l2} := \gamma_{\geq,0} \parallel \mathtt{l2}^{dep} := \gamma_{\mathsf{High},0} \parallel \mathtt{h} := \gamma_{\top_{val},0} \parallel \mathtt{h}^{dep} := \{\mathtt{h}\}$$

Note that we keep the precise value for `l1` and $\mathtt{l1}^{dep}$, as `l1` is not modified by the loop. The other variables may be changed and have to be abstracted. In particular `l2` may depend on `h` due to the implicit information-flow caused by the loop guard, which is reflected in the value for $\mathtt{l2}^{dep}$.

Applying now rule $\mathsf{invariantUpdate}^{\mathrm{dep}}$ and instantiating $\mathcal{U}'$ with $\mathcal{U}_{Inv}$ creates three new branches. For lack of space we focus on the third branch:

$$\mathtt{l1}^{dep} \doteq \{\mathtt{l1}\},\ \mathtt{l2}^{dep} \doteq \{\mathtt{l2}\},\ \mathtt{h}^{dep} \doteq \{\mathtt{h}\},\ \{\mathcal{U}_{Inv}\}\,!\mathtt{h} < 0$$
$$\Longrightarrow \{\mathcal{U}_{Inv}\}[\mathtt{C}](\mathtt{l1}^{dep} \dot{\subseteq} \{\mathtt{l1}\} \dot{\cup} \{\mathtt{l2}\})$$

Applying rule $\mathsf{ifElse}^{\mathrm{dep}}$ results in two branches. As we know that the conditional guard `l2<0` under $\mathcal{U}_{Inv}$ is never satisified, we can close the then-branch immediately. We continue on the else-branch and after a few rule applications and simplifications we are left with

$$\mathtt{l1}^{dep} \doteq \{\mathtt{l1}\},\ \mathtt{l2}^{dep} \doteq \{\mathtt{l2}\},\ \mathtt{h}^{dep} \doteq \{\mathtt{h}\},\ \gamma_{\top_{val},0} \geq 0 \Longrightarrow \{\} \dot{\subseteq} \{\mathtt{l1}\} \dot{\cup} \{\mathtt{l2}\}$$

This formula is obviously valid, and thus the program does not leak any information on the initial value of `h` to `l1`. Note that we would not have been able to prove that fact with a value-insensitive approach as we would then need to consider the possibility of the then branch injecting implicitly a High dependency into $\mathtt{l1}^{dep}$ via the conditional's guard. Note also that the security policy does not hold for program variable `l2`, and that the proof would not close if we had included $\mathtt{l2}^{dep} \dot{\subseteq} \{\mathtt{l1}\} \dot{\cup} \{\mathtt{l2}\}$ in our postcondition.

6 Related Work

Several approaches for combining deductive verification and abstract interpretation exist. One example is the "loop invariants on demand" technique [18], where an abstract interpretation system is invoked by a theorem prover to produce invariants for a specific program context. If the generated invariant is too weak, the abstract interpreter is iteratively called again using a more expressive abstract domain. Nevertheless, the theorem prover and the abstract interpreter are separate entities. In [19], a widening operator is built into a theorem prover.

Our goal of deeply integrating abstract interpretation into deductive verification based on dynamic logic is also pursued in [26]. There, the abstraction is done on logical formulas instead of on updates, using the technique of predicate abstraction [12]. The approach of [26] has not been applied to the problem of secure information flow.

For information flow analysis, our approach is more precise than typical security type systems, because it is flow- and value-sensitive. It is also more precise than the abstract interpretation defined in [11]. For example, in our setting not all locations to which a value is assigned in the body of a conditional or loop need depend on the guard (see Ex. 8).

Deductive approaches for reasoning about information-flow have been already listed in Sect. 1. Only some of the approaches focused on automation as one major concern. Papers [6] and [13] aim at the embedding of type-based analyses into program logics. To achieve full automation type-based systems are needed to construct either a certain formula entailing non-interference [6] or a derivation that can be translated into a proof of the program logic [13]. Neither includes a proof search algorithm.

The approach presented in [1] uses a Hoare logic and does not need a theorem prover to generate necessary invariants. On the other hand, it tracks only the independence relationship among variables and is therefore not value-sensitive. In [4] the authors use self-composition of programs to show information-flow security considering a formalisation of non-interference for a Hoare logic and an encoding in CTL. For the first one, automation is not targeted and for the second one model checking would be possible but is restricted to finite state programs.

7 Conclusion and Future Work

In this paper we presented a sound and relatively complete dynamic logic calculus that integrates abstract interpretation and keeps track of variable dependencies. The abstract domain is not fixed and the abstraction can be dynamically changed during symbolic execution. In the first part, we described an algorithm to compute loop invariants by abstraction-on-demand for a classical definition of a program logic. In the second part of the paper we extended the program logic to keep also track of variable dependencies so that information-flow can be modelled in a straightforward manner. We achieve the same degree of automation as type-based approaches while increasing the number of provable programs

due to value-sensitivity. The resulting calculus is close to the one used for JAVA in the KeY system [5] and we expect much of the machinery can be re-used.

In the future we want to focus on the followinng aspects: (i) extending the program logic to cover the sequential subset of JAVA; (ii) tracking dependencies even more precisely, e.g. currently an assignment such as `l = h-h` introduces `h` in the dependency set of `l`, even though it is a constant value in each program run and the symbolic execution machinery is in principle able to detect this. We have ideas how to treat such cases by extending our program logic semantics to include a trace semantics; (iii) supporting more sophisticated abstract interpretations involving infinite relational domains such as linear inequations; (iv) implementation and experimental evaluation including a comparison to other approaches.

References

1. Amtoft, T., Banerjee, A.: Information flow analysis in logical form. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, pp. 100–115. Springer, Heidelberg (2004)
2. Balser, M., Reif, W., Schellhorn, G., Stenzel, K., Thums, A.: Formal system development with KIV. In: Maibaum, T. (ed.) FASE 2000. LNCS, vol. 1783, p. 363–366. Springer, Heidelberg (2000)
3. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# Programming System: An Overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005)
4. Barthe, G., D'Argenio, P.R., Rezk, T.: Secure information flow by self-composition. In: 17th IEEE Computer Security Foundations Workshop, CSFW-17, Pacific Grove, CA, USA, pp. 100–114. IEEE Computer Society Press, Los Alamitos (2004)
5. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software. LNCS (LNAI), vol. 4334. Springer, Heidelberg (2007)
6. Beringer, L., Hofmann, M.: Secure information flow and program logics. In: 20th IEEE Computer Security Foundations Symposium CSF, Venice, Italy, pp. 233–248. IEEE Computer Society, Los Alamitos (2007)
7. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Fourth ACM Symposium on Principles of Programming Languages (POPL), Los Angeles, pp. 238–252. ACM Press, New York (1977)
8. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The ASTREÉ analyzer. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 21–30. Springer, Heidelberg (2005)
9. Darvas, Á., Hähnle, R., Sands, D.: A theorem proving approach to analysis of secure information flow. In: Hutter, D., Ullmann, M. (eds.) SPC 2005. LNCS, vol. 3450, pp. 193–209. Springer, Heidelberg (2005)
10. Filliâtre, J.-C., Marché, C.: The Why/Krakatoa/Caduceus platform for deductive program verification. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 173–177. Springer, Heidelberg (2007)
11. De Francesco, N., Martini, L.: Abstract interpretation to check secure information flow in programs with input-output security annotations. In: Dimitrakos, T., Martinelli, F., Ryan, P.Y.A., Schneider, S. (eds.) FAST 2005. LNCS, vol. 3866, pp. 63–80. Springer, Heidelberg (2006)

12. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
13. Hähnle, R., Pan, J., Rümmer, P., Walter, D.: Integration of a security type system into a program logic. Theoretical Computer Science 402(2-3), 172–189 (2008)
14. Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. Foundations of Computing. MIT Press, Cambridge (2000)
15. Holzmann, G.J.: The SPIN Model Checker. Pearson Education, London (2003)
16. Hunt, S., Sands, D.: On flow-sensitive security types. In: 33rd ACM Symposium on Principles of Programming Languages (POPL), pp. 79–90. ACM Press, New York (2006)
17. Joshi, R., Leino, K.R.M.: A semantic approach to secure information flow. Science of Computer Programming 37(1-3), 113–138 (2000)
18. Leino, K.R.M., Logozzo, F.: Loop invariants on demand. In: Yi, K. (ed.) APLAS 2005. LNCS, vol. 3780, pp. 119–134. Springer, Heidelberg (2005)
19. Leino, K.R.M., Logozzo, F.: Using widenings to infer loop invariants inside an SMT solver, or: A theorem prover as abstract domain. In: Proc. 1st International Workshop on Invariant Generation, WING 2007 (2007)
20. Nipkow, T., Paulson, L.C., Wenzel, M.T.: Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002)
21. Robby, M.B.D., Hatcliff, J.: Bogor: A flexible framework for creating software model checkers. In: McMinn, P. (ed.) Testing: Academia and Industry Conference; Practice And Research Techniques (TAIC PART), Windsor, United Kingdom, pp. 3–22. IEEE Computer Society, Los Alamitos (2006)
22. Rümmer, P.: Sequential, parallel, and quantified updates of first-order structures. In: Hermann, M., Voronkov, A. (eds.) LPAR 2006. LNCS (LNAI), vol. 4246, pp. 422–436. Springer, Heidelberg (2006)
23. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE Journal on Selected Areas in Communications 21(1), 5–19 (2003)
24. Velroyen, H., Rümmer, P.: Non-termination checking for imperative programs. In: Beckert, B., Hähnle, R. (eds.) TAP 2008. LNCS, vol. 4966, pp. 154–170. Springer, Heidelberg (2008)
25. Visser, W., Havelund, K., Brat, G.P., Park, S., Lerda, F.: Model checking programs. Automated Software Engineering 10(2), 203–232 (2003)
26. Weiß, B.: Predicate abstraction in a program logic calculus. In: Leuschel, M., Wehrheim, H. (eds.) IFM 2009. LNCS, vol. 5423, pp. 136–150. Springer, Heidelberg (2009)

A Formal Semantics

A.1 Basic Semantics

$$
\begin{aligned}
val_{I,s,\beta}(f(t_1,\ldots,t_n)) &= I(f)(val_{I,s,\beta}(t_1),\ldots,val_{I,s,\beta}(t_n)) \\
val_{I,s,\beta}(\mathbf{x}) &= s(\mathbf{x}) \\
val_{I,s,\beta}(y) &= \beta(y) \\
val_{I,s,\beta}(\mathit{if}(\varphi)\mathit{then}(t_1)\mathit{else}(t_2)) &= \begin{cases} val_{I,s,\beta}(t_1) & \text{if } val_{I,s,\beta}(\varphi) = tt \\ val_{I,s,\beta}(t_2) & \text{otherwise} \end{cases} \\
val_{I,s,\beta}(\{\mathcal{U}\}t) &= val_{I,s',\beta}(t) \qquad \text{where } s' = val_{I,s,\beta}(\mathcal{U})
\end{aligned}
$$

$$
\begin{aligned}
val_{I,s,\beta}(\text{true}) &= tt \\
val_{I,s,\beta}(\text{false}) &= ff \\
val_{I,s,\beta}(p(t_1,\ldots,t_n)) &= tt \quad \text{iff} \quad (val_{I,s,\beta}(t_1),\ldots,val_{I,s,\beta}(t_n)) \in I(p) \\
val_{I,s,\beta}(\varphi_1 \;\&\; \varphi_2) &= tt \quad \text{iff} \quad ff \notin \{val_{I,s,\beta}(\varphi_1), val_{I,s,\beta}(\varphi_2)\} \\
val_{I,s,\beta}(\varphi_1 \mid \varphi_2) &= tt \quad \text{iff} \quad tt \in \{val_{I,s,\beta}(\varphi_1), val_{I,s,\beta}(\varphi_2)\} \\
val_{I,s,\beta}(\varphi_1 \texttt{ -> } \varphi_2) &= val_{I,s,\beta}(\texttt{!}\,\varphi_1 \mid \varphi_2) \\
val_{I,s,\beta}(\texttt{!}\,\varphi) &= tt \quad \text{iff} \quad val_{I,s,\beta}(\varphi) = ff \\
val_{I,s,\beta}(\forall y.\varphi) &= tt \quad \text{iff} \quad ff \notin \{val_{I,s,\beta^v_y}(\varphi) \mid v \in \mathsf{D}\} \\
val_{I,s,\beta}(\exists y.\varphi) &= tt \quad \text{iff} \quad tt \in \{val_{I,s,\beta^v_y}(\varphi) \mid v \in \mathsf{D}\} \\
val_{I,s,\beta}(t_1 \doteq t_2) &= tt \quad \text{iff} \quad val_{I,s,\beta}(t_1) = val_{I,s,\beta}(t_2) \\
val_{I,s,\beta}(\{\mathcal{U}\}\varphi) &= val_{I,s',\beta}(\varphi) \qquad \text{where } s' = val_{I,s,\beta}(\mathcal{U}) \\
val_{I,s,\beta}([\mathtt{p}]\varphi) &= tt \quad \text{iff} \quad ff \notin \{val_{I,s',\beta}(\varphi) \mid s' \in val_{I,s,\beta}(\mathtt{p})\} \\
val_{I,s,\beta}(\mathtt{x}_1 := t_1 \parallel \ldots \parallel \mathtt{x}_n := t_n) &= \{\mathtt{x} \mapsto s(\mathtt{x}) \mid \mathtt{x} \notin \{\mathtt{x}_1,\ldots,\mathtt{x}_n\}\} \cup \\
&\quad \{\mathtt{x} \mapsto val_{I,s,\beta}(t_k) \mid \mathtt{x} = \mathtt{x}_k \text{ and } \mathtt{x} \notin \{\mathtt{x}_{k+1},\ldots,\mathtt{x}_n\}\} \\
val_{I,s,\beta}(\mathtt{x = t}) &= \{val_{I,s,\beta}(\mathtt{x} := t)\} \\
val_{I,s,\beta}(\mathtt{p1;p2}) &= \{val_{I,s',\beta}(\mathtt{p2}) \mid s' \in val_{I,s,\beta}(\mathtt{p1})\} \\
val_{I,s,\beta}(\mathtt{if(}g\mathtt{)\{p1\} else \{p2\}}) &= \begin{cases} val_{I,s,\beta}(\mathtt{p1}) & \text{if } val_{I,s,\beta}(g) = tt \\ val_{I,s,\beta}(\mathtt{p2}) & \text{otherwise} \end{cases} \\
val_{I,s,\beta}(\mathtt{while\ (}g\mathtt{)\ \{p\}}) &= \begin{cases} \bigcup_{s_1 \in S_1} val_{I,s_1,\beta}(\mathtt{while(}g\mathtt{)\ p}) & \text{if } val_{I,s,\beta}(g) = tt \\ \{s\} & \text{otherwise} \end{cases} \\
&\quad \text{where } S_1 = val_{I,s,\beta}(\mathtt{p})
\end{aligned}
$$

A.2 Semantics Enriched with Dependency Tracking

$$
\begin{aligned}
val'_{I,s,\beta}(\mathtt{x = t}) &= \{val_{I,s,\beta}(\mathtt{x} := t \parallel \mathtt{x}^{dep} := deps(t))\} \\
val'_{I,s,\beta}(\mathtt{p1;p2}) &= \{val'_{I,s',\beta}(\mathtt{p2}) \mid s' \in val'_{I,s,\beta}(\mathtt{p1})\} \\
val'_{I,s,\beta}(\mathtt{if(}g\mathtt{)\{p1\} else \{p2\}}) &= \begin{cases} S'_1 & \text{if } val_{I,s,\beta}(g) = tt \\ S'_2 & \text{otherwise} \end{cases} \\
&\quad \text{where } S_1 = val'_{I,s,\beta}(\mathtt{p1}), S_2 = val'_{I,s,\beta}(\mathtt{p2}), \\
&\quad S'_i = \emptyset \text{ iff } S_i = \emptyset, \text{otherwise } S'_i = \{s'_i\} \text{ where} \\
&\quad s'_i(\mathtt{x}) = \begin{cases} s_i(\mathtt{x}) & \text{if } \mathtt{x} \in \mathcal{PV} \text{ or} \\ & \mathtt{x} = \mathtt{y}^{dep} \text{ and} \\ & s_i(\mathtt{y}) = s(\mathtt{y}) \\ & \text{for } S_i = \{s_i\} \\ s_i(\mathtt{x}) \cup val_{I,s,\beta}(deps(g)) & \text{otherwise} \end{cases} \\
val'_{I,s,\beta}(\mathtt{while\ (}g\mathtt{)\ \{p\}}) &= \begin{cases} \bigcup_{s'_1 \in S'_1} val'_{I,s'_1,\beta}(\mathtt{while(}t\mathtt{)\ p}) & \text{if } val_{I,s,\beta}(g) = tt \\ \{s\} & \text{otherwise} \end{cases} \\
&\quad \text{where } S_1 = val'_{I,s,\beta}(\mathtt{p}), \\
&\quad \text{and where } S'_1 \text{ is derived from } S_1 \text{ as above}
\end{aligned}
$$

A.3 Dependencies of a Term or Formula

This section defines the function *deps* which takes a term or a formula (occuring inside a program) and returns a term that overapproximates the semantic dependencies of the argument. It is used both in the semantics with dependency tracking (App. A.2) and in the dependency-aware calculus rules (Sect. 5.2). Since logical variables, quantifiers, updates, nested programs, and dependency variables $\mathtt{x}^{dep} \in \mathcal{PV}^{dep}$ are not allowed to occur in programs, we refrain from providing a definition for these cases.

$$
\begin{aligned}
deps(f(t_1,\ldots,t_n)) &= deps(t_1) \dot\cup \ldots \dot\cup deps(t_n)\\
deps(\mathtt{x}) &= \mathtt{x}^{dep}\\
deps(\mathit{if}(\varphi)\mathit{then}(t_1)\mathit{else}(t_2)) &= deps(\varphi) \dot\cup deps(t_1) \dot\cup deps(t_2)\\
deps(a) &= \{\} \qquad \text{where } a \in \{\text{true}, \text{false}\}\\
deps(p(t_1,\ldots,t_n)) &= deps(t_1) \dot\cup \ldots \dot\cup deps(t_n)\\
deps(\varphi_1 * \varphi_2) &= deps(\varphi_1) \dot\cup deps(\varphi_2) \qquad \text{where } * \in \{\&, |, ->\}\\
deps(!\,\varphi) &= deps(\varphi)\\
deps(t_1 \doteq t_2) &= deps(t_1) \dot\cup deps(t_2)
\end{aligned}
$$

B Update Rewriting Rules

A rewrite rule $a \rightsquigarrow b$ is applicable to any occurrence of a within a sequent, and applying it means to replace that occurrence of a with b.

$$
\begin{aligned}
\{\mathcal{U}\}\{\mathtt{x}_1 := t_1 \parallel \ldots \parallel \mathtt{x}_n := t_n\} &\rightsquigarrow \{\mathcal{U} \parallel \mathtt{x}_1 := \{\mathcal{U}\}t_1 \parallel \ldots \parallel \mathtt{x}_n := \{\mathcal{U}\}t_n\}\\
\{\mathcal{U}\}f(t_1,\ldots,t_n) &\rightsquigarrow f(\{\mathcal{U}\}t_1,\ldots,\{\mathcal{U}\}t_n)\\
\{\mathtt{x}_1 := t_1 \parallel \ldots \parallel \mathtt{x}_n := t_n\}\mathtt{x} &\rightsquigarrow \begin{cases} \mathtt{x} & \text{if } \mathtt{x} \notin \{\mathtt{x}_1,\ldots,\mathtt{x}_n\}\\ t_k & \text{if } \mathtt{x} = \mathtt{x}_k \text{ and } \mathtt{x} \notin \{\mathtt{x}_{k+1},\ldots,\mathtt{x}_n\}\end{cases}\\
\{\mathcal{U}\}a &\rightsquigarrow a \qquad \text{where } a \in \mathcal{V} \cup \{\text{true}, \text{false}\}\\
\{\mathcal{U}\}\mathit{if}(\varphi)\mathit{then}(t_1)\mathit{else}(t_2) &\rightsquigarrow \mathit{if}(\{\mathcal{U}\}\varphi)\mathit{then}(\{\mathcal{U}\}t_1)\mathit{else}(\{\mathcal{U}\}t_2)\\
\{\mathcal{U}\}p(t_1,\ldots,t_n) &\rightsquigarrow p(\{\mathcal{U}\}t_1,\ldots,\{\mathcal{U}\}t_n)\\
\{\mathcal{U}\}(\varphi_1 * \varphi_2) &\rightsquigarrow \{\mathcal{U}\}\varphi_1 * \{\mathcal{U}\}\varphi_2 \qquad \text{where } * \in \{\&, |, ->\}\\
\{\mathcal{U}\}\,!\,\varphi &\rightsquigarrow !\{\mathcal{U}\}\varphi\\
\{\mathcal{U}\}\mathcal{Q}y.\varphi &\rightsquigarrow \mathcal{Q}y.\{\mathcal{U}\}\varphi \qquad \text{where } \mathcal{Q} \in \{\forall, \exists\}, y \notin \text{free}(\mathcal{U})\\
\{\mathcal{U}\}(t_1 \doteq t_2) &\rightsquigarrow \{\mathcal{U}\}t_1 \doteq \{\mathcal{U}\}t_2
\end{aligned}
$$

C Proofs

C.1 Lemma 1: Soundness of weakenUpdate

Proof. We assume that the following two statements hold for all I, s, β:

$$val_{I,s,\beta}\big(\Gamma, \{\mathcal{U}\}(\bar{\mathtt{x}} \doteq \bar{c}) \Longrightarrow \exists\bar{\gamma}.\{\mathcal{U}'\}(\bar{\mathtt{x}} \doteq \bar{c}), \Delta\big) = tt \tag{8}$$

$$val_{I,s,\beta}\big(\Gamma \Longrightarrow \{\mathcal{U}'\}\varphi, \Delta\big) = tt \tag{9}$$

Let I_0, s_0, β_0 be an arbitrary interpretation, state, and variable assignment. We need to show that $val_{I_0,s_0,\beta_0}(\Gamma \Longrightarrow \{\mathcal{U}\}\varphi, \Delta) = tt$. If $val_{I_0,s_0,\beta_0}(\bigwedge \Gamma) = ff$ or if $val_{I_0,s_0,\beta_0}(\bigvee \Delta) = tt$, then we are done immediately. Thus, we assume

$$val_{I_0,s_0,\beta_0}\big(\bigwedge(\Gamma \cup !\,\Delta)\big) = tt \tag{10}$$

and aim to prove that $val_{I_0,s_0,\beta_0}(\{\mathcal{U}\}\varphi) = tt$.

Let $s_1 = val_{I_0,s_0,\beta_0}(\mathcal{U})$, i.e., s_1 is the state reached by starting in s_0 and executing $\mathcal{U}$. Our goal is to prove that $val_{I_0,s_1,\beta}(\varphi) = tt$.

Let I_0' be the interpretation which is identical to I_0 except that $I_0'(\bar{c}) = s_1(\bar{\mathtt{x}})$, i.e., I_0' interprets the constant symbols $\bar{c}$ like the corresponding program variables $\bar{\mathtt{x}}$ are interpreted in s_1. This definition of I_0' implies $val_{I_0',s_0,\beta_0}(\bar{\mathtt{x}} \doteq \bar{c}) = tt$, and thus (as the symbols $\bar{c}$ do not occur in $\mathcal{U}$)

$$val_{I_0',s_0,\beta_0}\big(\{\mathcal{U}\}(\bar{\mathtt{x}} \doteq \bar{c})\big) = tt \tag{11}$$

Since the symbols $\bar{c}$ occur neither in Γ nor in Δ, and since I_0' is otherwise identical to I_0, we get from (10) that

$$val_{I_0',s_0,\beta_0}\big(\bigwedge(\Gamma \cup !\,\Delta)\big) = tt \tag{12}$$

Combining (12), (11) and the first premiss (8) yields

$$val_{I_0',s_0,\beta_0}\big(\exists\bar{\gamma}.\{\mathcal{U}'\}(\bar{\mathtt{x}} \doteq \bar{c})\big) = tt \tag{13}$$

This means that there is an interpretation I_0'' which is identical to I_0' except in the interpretation of the symbols $\bar{\gamma}$, and which satisfies

$$val_{I_0'',s_0,\beta_0}\big(\{\mathcal{U}'\}(\bar{\mathtt{x}} \doteq \bar{c})\big) = tt \tag{14}$$

Let $s_1' = val_{I_0'',s_0,\beta_0}(\mathcal{U}')$, i.e., s_1' is the state reached by starting in s_0 and executing $\mathcal{U}'$ under the interpretation I_0''. Equation 14 is equivalent to

$$val_{I_0'',s_1',\beta_0}(\bar{\mathtt{x}} \doteq \bar{c}) = tt \tag{15}$$

This means that $s_1'(\bar{\mathtt{x}}) = I_0''(\bar{c})$. Also, by definition of I_0'' and I_0', we have $I_0''(\bar{c}) = I_0'(\bar{c}) = s_1(\bar{\mathtt{x}})$. Thus, $s_1'(\bar{\mathtt{x}}) = s_1(\bar{\mathtt{x}})$, i.e., s_1' and s_1 are identical on all program variables $\bar{\mathtt{x}}$ which are potentially changed by either $\mathcal{U}$ or $\mathcal{U}'$. Since both s_1 and s_1' are derived from s_0 by executing one of these updates, this implies that $s_1' = s_1$. Inserting the definition of s_1', this reads as

$$val_{I_0'',s_0,\beta_0}(\mathcal{U}') = s_1 \tag{16}$$

Let I_1 be the interpretation identical to I_0'' except that the symbols $\bar{c}$ are interpreted as in I_0. Since the symbols $\bar{c}$ do not occur in $\mathcal{U}'$, we get from (16) that

$$val_{I_1,s_0,\beta_0}(\mathcal{U}') = s_1 \tag{17}$$

Since the $\bar{\gamma}$ do not occur in Γ nor in Δ, (10) tells us that

$$val_{I_1,s_0,\beta_0}\big(\bigwedge(\Gamma \cup\, !\,\Delta)\big) = tt \tag{18}$$

Combining (18) with the second premiss (9) yields

$$val_{I_1,s_0,\beta_0}\big(\{\mathcal{U}'\}\varphi\big) \tag{19}$$

With (17), this implies

$$val_{I_1,s_1,\beta_0}(\varphi) = tt \tag{20}$$

Since the symbols $\bar{\gamma}$ do not occur in φ, and since I_1 is otherwise identical to I_0, we get

$$val_{I_0,s_1,\beta_0}(\varphi) = tt \tag{21}$$

which is what we had to show. □

C.2 Lemma 2: Soundness of invariantUpdate

Proof. We assume that the following three statements hold for all I, s, β:

$$val_{I,s,\beta}\big(\Gamma, \{\mathcal{U}\}(\bar{\mathtt{x}} \doteq \bar{c}) \Longrightarrow \exists\bar{\gamma}.\{\mathcal{U}'\}(\bar{\mathtt{x}} \doteq \bar{c}), \Delta\big) = tt \tag{22}$$

$$val_{I,s,\beta}\big(\Gamma, \{\mathcal{U}'\}g,\ \{\mathcal{U}'\}[\mathtt{p}](\bar{\mathtt{x}} \doteq \bar{c}) \Longrightarrow \exists\bar{\gamma}.\{\mathcal{U}'\}(\bar{\mathtt{x}} \doteq \bar{c}), \Delta\big) = tt \tag{23}$$

$$val_{I,s,\beta}\big(\Gamma, \{\mathcal{U}'\}\,!\,g \Longrightarrow \{\mathcal{U}'\}[\ldots]\varphi, \Delta\big) = tt \tag{24}$$

Let I_0, s_0, β_0 be an arbitrary interpretation, state, and variable assignment. We need to show that $val_{I_0,s_0,\beta_0}\big(\Gamma \Longrightarrow \{\mathcal{U}\}[\texttt{while (}g\texttt{) \{p\}; }\ldots]\varphi, \Delta\big) = tt$. If $val_{I_0,s_0,\beta_0}\big(\bigwedge\Gamma\big) = ff$ or if $val_{I_0,s_0,\beta_0}\big(\bigvee\Delta\big) = tt$, then we are done immediately. Thus, we assume

$$val_{I_0,s_0,\beta_0}\big(\bigwedge(\Gamma \cup\, !\,\Delta)\big) = tt \tag{25}$$

and aim to prove that $val_{I_0,s_0,\beta_0}\big(\{\mathcal{U}\}[\texttt{while (}g\texttt{) \{p\}; }\ldots]\varphi\big) = tt$.

Let $s_1 = val_{I_0,s_0,\beta_0}(\mathcal{U})$, i.e., s_1 is the state reached by starting in s_0 and executing $\mathcal{U}$. If the loop does not terminate when started in s_1, then our proof goal $val_{I_0,s_0,\beta_0}\big(\{\mathcal{U}\}[\texttt{while (}g\texttt{) \{p\}; }\ldots]\varphi\big) = tt$ holds trivially. Therefore, we assume that the loop terminates. From the programming language semantics, we know that there is a finite sequence of states $s_1, \ldots, s_k$, where

$$val_{I_0,s_i,\beta_0}(\mathtt{p}) = \{s_{i+1}\} \qquad i \in \{1,\ldots,k-1\} \tag{26}$$

$$val_{I_0,s_i,\beta_0}(g) = tt \qquad i \in \{1,\ldots,k-1\} \tag{27}$$

$$val_{I_0,s_k,\beta_0}(g) = ff \tag{28}$$

Our task is to show that $val_{I_0,s_k,\beta_0}\big([\ldots]\varphi\big) = tt$.

We will use induction to prove that for all $i \in \{1, \ldots, k\}$, there is an interpretation I_i which is identical to I_0 except for the interpretation of the symbols $\bar{\gamma}$, and for which $val_{I_i,s_0,\beta_0}(\mathcal{U}') = s_i$. Intuitively, this means we show that for every state s_i of the chain, we can find an interpretation I_i of the symbols $\bar{\gamma}$ such that applying $\mathcal{U}'$ to the initial state s_0 with this interpretation I_i directly produces s_i. Afterwards, we will use this result and the third premiss (24) for showing $val_{I_0,s_k,\beta_0}\big([\ldots]\varphi\big) = tt$.

- *Base case* ($i = 1$). As our first premiss (22) is identical to the first premiss of the weakenUpdate rule (8), we can construct an interpretation I_1 with the desired properties in the same way as we did in the proof of updateWeaken (see (17)). For lack of space, we do not repeat this construction here.
- *Step case* ($i \in \{2, \ldots, k\}$). We assume that the induction hypothesis holds for $i-1$, i.e., there is an interpretation I_{i-1} identical to I_0 except for the interpretation of the symbols $\bar{\gamma}$, and which satisfies

$$val_{I_{i-1},s_0,\beta_0}(\mathcal{U}') = s_{i-1} \tag{29}$$

Let I'_{i-1} be the interpretation which is identical to I_{i-1} except that $I'_{i-1}(\bar{c}) = s_i(\bar{\mathtt{x}})$. This definition of I'_{i-1} implies $val_{I'_{i-1},s_i,\beta_0}(\bar{\mathtt{x}} \doteq \bar{c}) = tt$. As the symbols $\bar{c}$ do not occur in $\mathtt{p}$, we can combine this with (26) to get

$$val_{I'_{i-1},s_{i-1},\beta_0}\big([\mathtt{p}](\bar{\mathtt{x}} \doteq \bar{c})\big) = tt \tag{30}$$

By the induction hypothesis and the definition of I'_{i-1}, I'_{i-1} is identical to I_0 except in the interpretation of the symbols $\bar{\gamma}$ and $\bar{c}$. Since all of these occur neither in Γ nor in Δ, we get from (25) that

$$val_{I'_{i-1},s_0,\beta_0}\big(\bigwedge(\Gamma \cup !\,\Delta)\big) = tt \tag{31}$$

As the $\bar{c}$ do not occur in $\mathcal{U}'$, and as I'_{i-1} is otherwise identical to I_{i-1}, the induction hypothesis (29) also gives us

$$val_{I'_{i-1},s_0,\beta_0}(\mathcal{U}') = s_{i-1} \tag{32}$$

Together, (32) and (30) imply

$$val_{I'_{i-1},s_0,\beta_0}\big(\{\mathcal{U}'\}[\mathtt{p}](\bar{\mathtt{x}} \doteq \bar{c})\big) = tt \tag{33}$$

Since the symbols $\bar{\gamma}$ and $\bar{c}$ do not occur in g, we can combine (32) with (27) to get

$$val_{I'_{i-1},s_0,\beta_0}\big(\{\mathcal{U}'\}g\big) = tt \tag{34}$$

Taken together, (31), (34), (33) and the second premiss (23) yield

$$val_{I'_{i-1},s_0,\beta_0}\big(\exists\bar{\gamma}.\{\mathcal{U}'\}(\bar{\mathtt{x}} \doteq \bar{c})\big) = tt \tag{35}$$

This means that there is an interpretation I''_{i-1} which is identical to I'_{i-1} except in the interpretation of the symbols $\bar{\gamma}$, and which satisfies

$$val_{I''_{i-1},s_0,\beta_0}\big(\{\mathcal{U}'\}(\bar{\mathtt{x}} \doteq \bar{c})\big) = tt \tag{36}$$

Let $s'_i = val_{I''_{i-1},s_0,\beta_0}(\mathcal{U}')$, i.e., s_i is the state reached by starting in s_0 and executing $\mathcal{U}'$ under the interpretation I''_{i-1}. Equation (36) is equivalent to

$$val_{I''_{i-1},s'_i,\beta_0}(\bar{\mathtt{x}} \doteq \bar{c}) = tt \tag{37}$$

This means that $s'_i(\bar{\mathtt{x}}) = I''_{i-1}(\bar{c})$. Also, by definition of I''_{i-1} we have $I''_{i-1}(\bar{c}) = s_i(\bar{\mathtt{x}})$. Thus $s'_i = s_i$. Inserting the definition of s'_i, this reads as

$$val_{I''_{i-1},s_0,\beta_0}(\mathcal{U}') = s_i \tag{38}$$

Let I_i be the interpretation identical to I''_{i-1} except that the symbols $\bar{c}$ are interpreted as in I_{i-1}. Since the $\bar{c}$ do not occur in $\mathcal{U}'$, we get from (38) that

$$val_{I_i,s_0,\beta_0}(\mathcal{U}') = s_i \tag{39}$$

Since I_i also differs from I_0 only in the interpretation of the symbols $\bar{\gamma}$, it has both desired properties.

This finishes our induction. We know now that in particular for $i = k$, there is an interpretation I_k which is identical to I_0 except for the interpretation of the symbols $\bar{\gamma}$, and for which

$$val_{I_k,s_0,\beta_0}(\mathcal{U}') = s_k \tag{40}$$

Since the symbols $\bar{\gamma}$ do not occur in g, we can combine this with (28) to get

$$val_{I_k,s_0,\beta_0}\big(\{\mathcal{U}'\}g\big) = ff \tag{41}$$

Since the $\bar{\gamma}$ also do not occur in Γ nor Δ, (25) tells us that

$$val_{I_k,s_0,\beta_0}\big(\bigwedge(\Gamma \cup !\,\Delta)\big) = tt \tag{42}$$

Combining (42) and (41) with the third premiss (24) yields

$$val_{I_k,s_0,\beta_0}\big(\{\mathcal{U}'\}[\ldots]\varphi\big) \tag{43}$$

With (40), this implies

$$val_{I_k,s_k,\beta_0}\big([\ldots]\varphi\big) = tt \tag{44}$$

Since the symbols $\bar{\gamma}$ do not occur in $[\ldots]\varphi$, and since I_k is otherwise identical to I_0, we get

$$val_{I_0,s_k,\beta_0}\big([\ldots]\varphi\big) = tt \tag{45}$$

which is what we had to show. □