Program Transformation Based on Symbolic Execution and Deduction (Technical Report)*

Ran Ji and Reiner Hähnle and Richard Bubel

Department of Computer Science
Technische Universität Darmstadt, Germany
{ran,haehnle,bubel}@cs.tu-darmstadt.de

Abstract. We present a program transformation framework based on symbolic execution and deduction. Its virtues are: (i) behavior preservation of the transformed program is guaranteed by a sound program logic, and (ii) automated first-order solvers are used for simplification and optimization. Transformation consists of two phases: first the source program is symbolically executed by sequent calculus rules in a program logic. This involves a precise analysis of variable dependencies, aliasing, and elimination of infeasible execution paths. In the second phase, the target program is synthesized by a leaves-to-root traversal of the symbolic execution tree by backward application of (extended) sequent calculus rules. We prove soundness by a suitable notion of bisimulation and we discuss one possible approach to automated program optimization.

1 Introduction

State-of-the-art program verification systems can show the correctness of complex software written in industrial programming languages [1]. The main reason why functional verification is not used routinely is that considerable expertise is required to come up with formal specifications [2], invariants, and proof hints. Nevertheless, modern software verification systems are an impressive achievement: they contain a fully formal semantics of industrial programming languages and, due to automated first-order reasoning and highly developed heuristics, in fact a high degree of automation is achieved: more than 99,9% of the proof steps are typically completely automatic. Given the right annotations and contracts, often 100% automation is possible. This paper is about leveraging the enormous potential of verification tools that at the moment goes unused.

The central observation is that everything making functional verification hard, is in fact not needed if one is mainly interested in simplifying and optimizing a program rather than proving it correct. First, there is no need for complex formal specifications: the property that two programs are bisimilar on observable locations is easy to express schematically. Second, complex invariants

* This work has been partially supported by the IST program of the European Commission, Future and Emerging Technologies under the IST-231620 HATS project.

are only required to prove non-trivial postconditions. If the preservation of behavior becomes the only property to be proven, then simple, schematic invariants will do. Hence, complex formulas are absent, which does away with the need for difficult quantifier instantiations.

On the other hand, standard verification tools are not set up to relate a source and a target program, which is what is needed for program simplification and optimization. The main contribution of this paper is to adapt the program logic of a state-of-the-art program verifier [3] to the task of sound program transformation and to show that fully automatic program simplification and optimization with guaranteed soundness is possible as a consequence.

This paper extends previous work [4], where the idea of program specialization via a verification tool was presented for the first time. We remodeled the ad-hoc semantics of the earlier paper in terms of standard bisimulation theory [5]. While this greatly improves the presentation, more importantly, it enables the new optimization described in Section 6.

Aiming at a concise presentation, we employ the small OO imperative programming language PL. It contains essential features of OO languages, but abstracts away from technicalities that complicate the presentation. Section 2 introduces PL and Section 3 defines a program logic for it with semantics and a calculus. These are adapted to the requirements of program transformation in Section 4. In Section 6 we harvest from our effort and add a non-trivial optimization strategy. We close with related work (Section 7) and future work (Section 8).

2 Programming Language

PL supports classes, objects, attributes, method polymorphism (but not method overloading). Unsupported features are generic types, exceptions, multi-threading, floating points, and garbage collection. The types of PL are the types derived from class declarations, the type `int` of mathematical integers ($\mathbb{Z}$), and the standard Boolean type `boolean`.

A PL program `p` is a non-empty set of class declarations, where each class defines a class type. PL contains at least two class types `Object` and `Null`. The class hierarchy (without `Null`) forms a tree with class `Object` as root. The type `Null` is a singleton with `null` as its only element and may be used in place of any class type. It is the smallest class type.

A class $Cl := (cname, scname_{opt}, fld, mtd)$ consists of (i) a classname $cname$ unique in `p`, (ii) the name of its superclass $scname$ (optional, only omitted for $cname =$ `Object`), (iii) a list of field declarations fld and method declarations mtd. The syntax coincides with that of Java. The only features lacking from Java are constructors and initialization blocks. We use some conventions: if not stated otherwise, any sequence of statements is viewed as if it were the body of a static, void method declared in a class `Default` with no fields.

The syntax of the executable fragment of PL is given in Fig. 1.

Statements

$stmnt ::= stmnt\ \ stmnt \mid lvarDecl \mid locExp\texttt{'='}exp\texttt{';'} \mid cond \mid loop$

$loop ::= \texttt{while '('}exp\texttt{')' '\{'}stmnt\texttt{'\}'}$

$lvarDecl ::= Type\ \ \texttt{IDENT ('='}\ exp)_{opt}\texttt{';'}$

$cond ::= \texttt{if '('}exp\texttt{')' '\{'}stmnt\texttt{'\}' else '\{'}stmnt\texttt{'\}'}$

Expressions

$exp ::= \texttt{(}exp\texttt{.)}_{opt}\, mthdCall \mid opExp \mid locExp$

$mthdCall ::= \texttt{mthdName'('}exp_{opt}\texttt{(','}exp\texttt{)*')'}$

$opExp ::= opr\texttt{(}exp_{opt}\texttt{(,}exp\texttt{)*)} \mid \mathbb{Z} \mid \texttt{TRUE} \mid \texttt{FALSE} \mid \texttt{null}$

$opr ::= \texttt{!} \mid \texttt{-} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>=} \mid \texttt{>} \mid \texttt{==} \mid \texttt{\&\&} \mid \texttt{||} \mid \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%} \mid \texttt{++}$

Locations

$locExp ::= \texttt{IDENT} \mid exp\texttt{.IDENT}$

Fig. 1. Syntax of PL.

Any complex statement can be easily decomposed into a sequence of simpler statements without changing the meaning of a program, e.g., `y = z ++;` can be decomposed into `int t = z; z = z + 1; y = t;`, where `t` is a *fresh* variable, not used anywhere else. As we shall see later, a suitable notion of simplicity is essential, for example, to compute variable dependencies and simplify symbolic states. This is built into our semantics and calculus, so we need a precise definition of *simple statements*. In Fig. 2, statements in the syntactic category *spStmnt* have at most one source of side effect each. This can be a non-terminating expression (such as a null pointer access), a method call, or an assignment to a location.

$spStmnt ::= spLvarDecl \mid locVar\texttt{'='}spExp\texttt{';'} \mid locVar\texttt{'='}spAtr\texttt{';'} \mid spAtr\texttt{'='}spExp\texttt{';'}$

$spLvarDecl ::= Type\ \ \texttt{IDENT';'}$

$spExp ::= \texttt{(}locVar\texttt{.)}_{opt}\, spMthdCall \mid spOpExp \mid litVar$

$spMthdCall ::= \texttt{mthdName'('}litVar_{opt}\texttt{(','}litVar\texttt{)*')'}$

$spOpExp ::= \texttt{!}litVar \mid \texttt{-}litVar \mid litVar\ \ binOpr\ \ litVar$

$litVar ::= litval \mid locVar \qquad litval ::= \mathbb{Z} \mid \texttt{TRUE} \mid \texttt{FALSE} \mid \texttt{null}$

$binOpr ::= \texttt{<} \mid \texttt{<=} \mid \texttt{>=} \mid \texttt{>} \mid \texttt{==} \mid \texttt{\&\&} \mid \texttt{||} \mid \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%}$

$locVar ::= \texttt{IDENT}$

$spAtr ::= locVar\texttt{.IDENT}$

Fig. 2. Syntax of PL simple statements.

3 Program Logic and Sequent Calculus

Symbolic execution was introduced independently by King [6] and others in the early 1970s. The main idea is to take symbolic values (terms) instead of concrete ones for the initial values of input variables, fields, etc., for program execution.

The interpreter then performs algebraic computations on terms instead of computing concrete results. In this paper, following [7], symbolic execution is done by applying *sequent calculus* rules of a program logic. Sequent calculi are often used to verify a program against a specification [7], but here we focus on symbolic execution, which we embed into a program logic for the purpose of being able to argue the correctness of program transformations and optimizations.

3.1 Program Logic

Our program logic is *dynamic logic (DL)* [8]. The target program occurs in unencoded form as a first-class citizen inside the logic's connectives. Sorted first-order dynamic logic is sorted first-order logic that is syntactically closed wrt program correctness modalities $[\cdot]\cdot$ (box) and $\langle\cdot\rangle\cdot$ (diamond). The first argument is a program and the second a dynamic logic formula. Let $\mathtt{p}$ denote a program and ϕ a dynamic logic formula then $[\mathtt{p}]\phi$ and $\langle\mathtt{p}\rangle\phi$ are DL-formulas. Informally, the former expresses that if $\mathtt{p}$ is executed and terminates *then* in all reached final states ϕ holds; the latter means that if $\mathtt{p}$ is executed then it terminates *and* in at least one of the reached final states ϕ holds.

We consider only deterministic programs, hence, a program $\mathtt{p}$ executed in a given state s *either* terminates and reaches exactly *one* final state *or* it does not terminate and there are no reachable final states. The box modality expresses *partial correctness* of a program, while the diamond modality coincides with *total correctness*.

A dynamic logic based on PL-programs is called PL-DL. The signature of the program logic depends on a *context* PL*-program* $\mathcal{C}$.

Definition 1 (PL-Signature $\Sigma_\mathcal{C}$). *A* signature $\Sigma_\mathcal{C} = (\mathsf{Srt}, \preceq, \mathsf{Pred}, \mathsf{Func}, \mathsf{LgV})$ *consists of:*

(i) *a set of names* Srt *called* sorts *containing at least one sort for each primitive type and one for each class Cl declared in $\mathcal{C}$:* $\mathsf{Srt} \supseteq \{\mathtt{int}, \mathtt{boolean}\} \cup \{Cl \mid \textit{for all classes } Cl \textit{ declared in } \mathcal{C}\}$*;*

(ii) *a* partial subtyping order $\preceq\colon \mathsf{Srt} \times \mathsf{Srt}$ *that models the subtype hierarchy of $\mathcal{C}$ faithfully;*

(iii) *a set of* predicate symbols $\mathsf{Pred} := \{p : T_1 \times \ldots \times T_n \mid T_i \in \mathsf{Srt}, n \in \mathbb{N}\}$*. We call $\alpha(p) = T_1 \times \ldots \times T_n$ the* signature *of the predicate symbol.*

(iv) *a set of* function symbols $\mathsf{Func} := \{f : T_1 \times \ldots \times T_n \to T \mid T_i, T \in \mathsf{Srt}, n \in \mathbb{N}\}$*. We call $\alpha(f) = T_1 \times \ldots \times T_n \to T$ the* signature *of the function symbol.* $\mathsf{Func} := \mathsf{Func}_r \cup \mathsf{PV} \cup \mathsf{Attr}$ *is further divided into disjoint subsets:*

- *the* rigid *function symbols* Func_r*;*
- *the* program variables $\mathsf{PV} = \{\mathtt{i}, \mathtt{j}, \ldots\}$*, which are* non-rigid *constants;*
- *the* non-rigid *function symbols* attribute Attr*, such that for each attribute* $\mathtt{a}$ *of type T declared in class Cl an attribute function $\mathtt{a}@Cl : Cl \to T \in \mathsf{Attr}$ exists. We omit the $@C$ from attribute function names if no ambiguity arises.*

(v) *a set of* logical variables $\mathsf{LgV} := \{x : T \,|\, T \in \mathsf{Srt}\}$.

We distinguish between *rigid* and *non-rigid* function and predicate symbols. Intuitively, the semantics of rigid symbols does not depend on the current state of program execution, while non-rigid symbols are state-dependent. Local program variables, static, and instance fields are modeled as non-rigid function symbols and together form a separate class of non-rigid symbols called *location* symbols. Specifically, local program variables and static fields are modeled as non-rigid constants, instance fields as unary non-rigid functions.

$\Pi_{\Sigma_\mathcal{C}}$ denotes the set of all executable PL programs (i.e., sequences of statements) with locations over signature $\Sigma_\mathcal{C}$. In the remaining paper, we use the notion of a program to refer to a sequence of executable PL-statements. If we want to include class, interface or method declarations, we either include them explicitly or make a reference to the context program $\mathcal{C}$.

The inductive definition of terms and formulas is standard, but we introduce a new syntactic category called *update* to represent state updates with symbolic expressions.

Definition 2 (Terms, Updates and Formulas). Terms t, updates u *and* formulas ϕ *are* well-sorted *first-order expressions of the following kind:*

$$
\begin{aligned}
t &::= x \mid \mathtt{i} \mid t.\mathtt{a} \mid f(t,\ldots,t) \mid (\phi\ ?\ t\ :\ t) \mid \mathbb{Z} \mid \mathtt{TRUE} \mid \mathtt{FALSE} \mid \mathtt{null} \mid \{u\}t \\
u &::= \mathtt{i} := t \mid t.\mathtt{a} := t \mid u \parallel u \mid \{u\}u \\
\phi &::= true \mid false \mid p(t,\ldots,t) \mid \neg\phi \mid \phi \circ \phi\ (\circ \in \{\wedge,\vee,\rightarrow,\leftrightarrow\}) \mid (\phi\ ?\ \phi\ :\ \phi) \mid \\
&\quad \forall x:T.\phi \mid \exists x:T.\phi \mid [\mathtt{p}]\phi \mid \langle\mathtt{p}\rangle\phi \mid \{u\}\phi
\end{aligned}
$$

where $\mathtt{a} \in \mathsf{Attr}, f \in \mathsf{Func}, p \in \mathsf{Pred}, \mathtt{i} \in \mathsf{PV}, x:T \in \mathsf{LgV}$*, and* $\mathtt{p}$ *is a sequence of executable* PL *statements.*

An *elementary update* $\mathtt{i} := t$ or $t.\mathtt{a} := t$ is a pair of location and term. They are of *single static assignment (SSA)* form [9,10], with the same meaning as simple assignments. Elementary updates are composed to *parallel updates* $u_1 \| u_2$ and work like simultaneous assignments. Updates applied to terms or formulas are again terms or formulas.

Terms, formulas and updates are evaluated with respect to a PL-DL Kripke structure.

Definition 3 (Kripke structure). *A* PL*-DL Kripke structure* $\mathcal{K}_{\Sigma_{\mathrm{PL}}} = (\mathcal{D}, I, S)$ *consists of*

(i) a set of elements $\mathcal{D}$ *called domain,*
(ii) an interpretation I *with*
- $I(T) = \mathcal{D}_T,\ T \in \mathsf{Srt}$ *assigning each sort its non-empty domain* $\mathcal{D}_T$*. It adheres to the restrictions imposed by the subtype order* $\preceq$*;* $\mathtt{Null}$ *is always interpreted as a singleton set and subtype of all class types;*
- $I(f) : \mathcal{D}_{T_1} \times \ldots \times \mathcal{D}_{T_n} \rightarrow \mathcal{D}_T$ *for each rigid function symbol* $f : T_1 \times \ldots \times T_n \rightarrow T \in \mathsf{Func}_r$*;*
- $I(p) \subseteq \mathcal{D}_{T_1} \times \ldots \times \mathcal{D}_{T_n}$ *for each predicate symbol* $p : T_1 \times \ldots \times T_n \in \mathsf{Pred}$*;*

(iii) a set of states S assigning meaning to non-rigid function symbols: let $s \in S$ then $s(\mathtt{a@}Cl) : \mathcal{D}_{Cl} \to \mathcal{D}_T$, $\mathtt{a@}Cl : Cl \to T \in \mathsf{Attr}$ and $s(\mathtt{i}) : \mathcal{D}_T$, $\mathtt{i} \in \mathsf{PV}$.

The pair $D = (\mathcal{D}, I)$ is called a first-order structure.

As usual in first-order logic, to define evaluation of terms and formulas in addition to a structure we need the notion of a *variable assignment*. A *variable assignment* $\beta : \mathsf{LgV} \to \mathcal{D}_T$ maps a logical variable $x : T$ to its domain $\mathcal{D}_T$.

Definition 4 (Evaluation function). *A term, formula or update is evaluated relative to a given first-order structure $D = (\mathcal{D}, I)$, a state $s \in S$ and a variable assignment β, while programs and expressions are evaluated relative to a D and $s \in S$. The* evaluation function *val is defined recursively. It evaluates*

(i) every term $t : T$ to a value $val_{D,s,\beta}(t) \in \mathcal{D}_T$;
(ii) every formula ϕ to a truth value $val_{D,s,\beta}(\phi) \in \{tt, ff\}$;
(iii) every update u to a state transformer $val_{D,s,\beta}(u) \in S \to S$;
(iv) every expression $e : T$ to a set of pairs of state and value $val_{D,s}(e) \subseteq 2^{S \times T}$;
(v) every statement st to a set of states $val_{D,s}(st) \subseteq 2^S$.

Since PL is deterministic, all sets of states or state-value pairs have at most one element.

Fig. 3 shows a collection of the semantic definition. The expression $s[\mathtt{x} \leftarrow \mathtt{v}]$ denotes a state coincides with s except at $\mathtt{x}$ which is mapped to the evaluation of $\mathtt{v}$.

Example 1 (Update semantics). We illustrate the semantics of updates of Fig. 3. Evaluating $\{\mathtt{i} := \mathtt{j} + 1\}\mathtt{i} \geq \mathtt{j}$ in a state s is identical to evaluating the formula $\mathtt{i} \geq \mathtt{j}$ in a state s' which coincides with s except for the value of $\mathtt{i}$ which is evaluated to the value of $val_{D,s,\beta}(\mathtt{j} + 1)$. Evaluation of the parallel update $\mathtt{i} := \mathtt{j} \| \mathtt{j} := \mathtt{i}$ in a state s leads to the successor state s' identical to s except that the values of $\mathtt{i}$ and $\mathtt{j}$ are swapped. The parallel update $\mathtt{i} := 3 \| \mathtt{i} := 4$ has a *conflict* as $\mathtt{i}$ is assigned different values. In such a case the last occurring assignment $\mathtt{i} := 4$ overrides all previous ones of the same location. Evaluation of $\{\mathtt{i} := \mathtt{j}\}\{\mathtt{j} := \mathtt{i}\}\phi$ in a state s results in evaluating ϕ in a state, where $\mathtt{i}$ has the value of $\mathtt{j}$, and $\mathtt{j}$ remains unchanged.

Remark. $\{\mathtt{i} := \mathtt{j}\}\{\mathtt{j} := \mathtt{i}\}\phi$ is the sequential application of updates $\mathtt{i} := \mathtt{j}$ and $\mathtt{j} := \mathtt{i}$ on the formula ϕ. To ease the presentation, we overload the concept of update and also call $\{\mathtt{i} := \mathtt{j}\}\{\mathtt{j} := \mathtt{i}\}$ an update. In the following context, if not stated otherwise, we use the upper-case letter $\mathcal{U}$ to denote this kind of update, compared to the real update that is denoted by a lower-case letter u. An update $\mathcal{U}$ could be the of form $\{u\}$ and $\{u_1\} \ldots \{u_n\}$. Furthermore, $\{u_1\} \ldots \{u_n\}$ can be simplified into the form of $\{u\}$, namely the *normal form* (NF) of update.

Definition 5 (Normal form of update). *An update is in* normal form*, denoted by $\mathcal{U}^{nf}$, if it has the shape $\{u_1 \| \ldots \| u_n\}$, $n \geq 0$, where each u_i is an elementary update and there is no conflict between u_i and u_j for any $i \neq j$.*

For terms:

$val_{D,s,\beta}(\texttt{TRUE}) = \mathit{True}$

$val_{D,s,\beta}(\texttt{FALSE}) = \mathit{False},$ where $\{\mathit{True}, \mathit{False}\} = D(\texttt{boolean})$

$val_{D,s,\beta}(x) = \beta(x),\ x \in \mathsf{LgV}$

$val_{D,s,\beta}(\mathtt{x}) = s(\mathtt{x}),\ \mathtt{x} \in \mathsf{PV}$

$val_{D,s,\beta}(o.\mathtt{a}) = s(\mathtt{a})(val_{D,s,\beta}(o)),\ \mathtt{a} \in \mathsf{Attr}$

$val_{D,s,\beta}(f(t_1, \ldots, t_n)) = D(f)(val_{D,s,\beta}(t_1), \ldots, val_{D,s,\beta}(t_n))$

$$val_{D,s,\beta}(\psi\ ?\ t_1\ :\ t_2) = \begin{cases} val_{D,s,\beta}(t_1) \text{ if } val_{D,s,\beta}(\psi) = tt \\ val_{D,s,\beta}(t_2) \text{ otherwise} \end{cases}$$

$val_{D,s,\beta}(\{u\}t) = val_{D,s',\beta}(t),\ s' = val_{D,s,\beta}(u)(s)$

For formulas:

$val_{D,s,\beta}(true) = tt$

$val_{D,s,\beta}(false) = ff$

$val_{D,s,\beta}(p(t_1, \ldots, t_n)) = tt$ iff $(val_{D,s,\beta}(t_1), \ldots, val_{D,s,\beta}(t_n)) \in D(p)$

$val_{D,s,\beta}(\neg\phi) = tt$ iff $val_{D,s,\beta}(\phi) = ff$

$val_{D,s,\beta}(\psi \wedge \phi) = tt$ iff $val_{D,s,\beta}(\psi) = tt$ and $val_{D,s,\beta}(\psi) = tt$

$val_{D,s,\beta}(\psi \vee \phi) = tt$ iff $val_{D,s,\beta}(\psi) = tt$ or $val_{D,s,\beta}(\psi) = tt$

$val_{D,s,\beta}(\psi \rightarrow \phi) = val_{D,s,\beta}(\neg\psi \vee \phi)$

$val_{D,s,\beta}(\psi \leftrightarrow \phi) = val_{D,s,\beta}(\psi \rightarrow \phi \wedge \phi \rightarrow \psi)$

$val_{D,s,\beta}([\mathtt{p}]\phi) = tt$ iff $ff \notin \{val_{D,s',\beta}(\phi) | s' \in val_{D,s}(\mathtt{p})\}$

$val_{D,s,\beta}(\{u\}\phi) = val_{D,s',\beta}(\phi),$ where $s' = val_{D,s,\beta}(u)(s)$

For updates:

$val_{D,s,\beta}(\mathtt{x} := t)(s) = s[\mathtt{x} \leftarrow t]$

$val_{D,s,\beta}(o.\mathtt{a} := t)(s) = s[(\mathtt{a})(val_{D,s,\beta}(o)) \leftarrow t]$

$val_{D,s,\beta}(u_1 \| u_2)(s) = val_{D,s,\beta}(u_2)(val_{D,s,\beta}(u_1)(s))$

$val_{D,s,\beta}(\{u_1\}u_2)(s) = val_{D,s',\beta}(u_2)(s'),$ where $s' = val_{D,s,\beta}(u_1)(s)$

For expressions:

$val_{D,s}(\mathtt{x}) = \{(s, s(\mathtt{x}))\},\ \mathtt{x} \in \mathsf{PV}$

$val_{D,s}(o.\mathtt{a}) = \{(s', s(a)(d)) \mid (s', d) \in val_{D,s}(o) \wedge d \neq null\}$

$val_{D,s}(e_1 \circ e_2) = \{(s'', D(\circ)(d_1, d_2)) \mid (s', d_1) \in val_{D,s}(e_1) \wedge (s'', d_2) \in val_{D,s'}(e_2)\}$
$\circ \in \{+, -, *, \ldots\}$

For statements:

$val_{D,s}(\mathtt{x} = e) = \{s'[\mathtt{x} \leftarrow d] \mid (s', d) \in val_{D,s}(e)\},\ \mathtt{x} \in \mathsf{PV}$

$val_{D,s}(o.\mathtt{a} = e) = \{s''[\mathtt{a}(d_o) \leftarrow d_e] \mid (s', d_o) \in val_{D,s}(o) \wedge (s'', d_e) \in val_{D,s'}(e)\}$

$val_{D,s}(\mathtt{p_1}; \mathtt{p_2}) = \bigcup_{s' \in val_{D,s}(\mathtt{p_1})} val_{D,s'}(\mathtt{p_2})$

$$val_{D,s}(\mathtt{if}(e)\ \{\mathtt{p}\}\ \mathtt{else}\ \{\mathtt{q}\}) = \begin{cases} val_{D,s',\beta}(\mathtt{p}), & (s', \mathit{True}) \in val_{D,s}(e) \\ val_{D,s',\beta}(\mathtt{q}), & (s', \mathit{False}) \in val_{D,s}(e) \\ \emptyset, & \text{otherwise} \end{cases}$$

$$val_{D,s}(\mathtt{while}(e)\{\mathtt{p}\}) = \begin{cases} \bigcup_{s_1 \in S_1} val_{D,s_1}(\mathtt{while}(e)\{\mathtt{p}\}) \text{ where } S_1 = val_{D,s'}(\mathtt{p}), \\ \qquad \text{if } (s', \mathit{True}) \in val_{D,s}(e) \\ \{s'\},\ \text{if } (s', \mathit{False}) \in val_{D,s}(e) \\ \emptyset,\ \text{otherwise} \end{cases}$$

Fig. 3. Definition of PL-DL semantic evaluation function.

Example 2 (Normal form of update). For the following updates,

- $\{i := j + 1\}$ and $\{i := j + 1 \| j := i\}$ are in normal form.
- $\{i := j + 1\}\{j := i\}$ is not in normal form.
- $\{i := j+1 \| j := i \| i := i+1\}$ is not in normal form, because there is a conflict between $i := j + 1$ and $i := i + 1$.

The normal form of an update $\mathcal{U} = \{u_1\} \ldots \{u_n\}$ can be achieved by applying a sequence of *update simplification* steps shown in Fig. 4. Soundness of these rules and that they achieve normal form are proven in [11].

$$\{\ldots \| \mathtt{x} := v_1 \| \ldots \| \mathtt{x} := v_2 \| \ldots\} v \rightsquigarrow \{\ldots \| \ldots \| \ldots \| \mathtt{x} := v_2 \| \ldots\} v \quad \text{where } v \in t \cup f \cup \phi$$

$$\{\ldots \| \mathtt{x} := v' \| \ldots\} v \rightsquigarrow \{\ldots \| \ldots\} v, \text{ where } v \in t \cup f \cup \phi,\ \mathtt{x} \notin fpv(v)$$

$$\{u\}\{u'\} v \rightsquigarrow \{u \| \{u\} u'\} v, \text{ where } v \in t \cup f \cup \phi$$

$$\{u\} x \rightsquigarrow x, \text{ where } x \in \mathsf{LgV}$$

$$\{u\} f(t_1, \ldots, t_n) \rightsquigarrow f(\{u\}(t_1), \ldots, \{u\}(t_n))$$

$$\{u\} \neg \phi \rightsquigarrow \neg \{u\} \phi$$

$$\{u\}(\phi_1 \circ \phi_2) \rightsquigarrow \{u\}(\phi_1) \circ \{u\}(\phi_2), \text{ where } \circ \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$$

$$\{u\}(\mathtt{x} := v) \rightsquigarrow \mathtt{x} := \{u\} v$$

$$\{u\}(o.\mathtt{a} := v) \rightsquigarrow o.\mathtt{a} := \{u\} v$$

$$\{u\}(u_1 \| u_2) \rightsquigarrow \{u\} u_1 \| \{u\} u_2$$

$$\{\mathtt{x} := v\}\mathtt{x} \rightsquigarrow v$$

$$\{o.\mathtt{a} := v\} o.\mathtt{a} \rightsquigarrow v$$

Fig. 4. Update simplification rules.

Finally, we give the definitions of satisfiability, model and validity of formulas.

Definition 6 (Satisfiability, model and validity). *A formula ϕ*

- *is* satisfiable*, denoted by $D, s, \beta \models \phi$, if there exists a first-order structure D, a state $s \in S$ and a variable assignment β with $val_{D,s,\beta}(\phi) = tt$.*
- *has a* model*, denoted by $D, s \models \phi$, if there exists a first-order structure D, a state $s \in S$, such that for all variable assignments β: $val_{D,s,\beta}(\phi) = tt$ holds.*
- *is* valid*, denoted by $\models \phi$, if for all first-order structures D, states $s \in S$ and for all variable assignments β: $val_{D,s,\beta}(\phi) = tt$ holds.*

3.2 Sequent Calculus

We define a sequent calculus for PL-DL. Symbolic execution of a PL-program is performed by application of sequent calculus rules. Soundness of the rules ensures validity of provable PL-DL formulas in a program verification setting [3].

A *sequent* is a pair of sets of formulas $\Gamma = \{\phi_1, \ldots, \phi_n\}$ (antecedent) and $\Delta = \{\psi_1, \ldots, \psi_m\}$ (succedent) of the form $\Gamma \Longrightarrow \Delta$. Its semantics is defined by the formula $\bigwedge_{\phi \in \Gamma} \phi \rightarrow \bigvee_{\psi \in \Delta} \psi$. A *sequent calculus rule* has one conclusion and zero or more premises. It is applied to a sequent s by matching its conclusion against s. The instantiated premises are then added as children of s. Our PL-DL sequent calculus behaves as a symbolic interpreter for PL. A *sequent* for PL-DL is always of the form $\Gamma \Longrightarrow \mathcal{U}[\mathtt{p}]\phi, \Delta$. During symbolic execution performed by the sequent rules (see Fig. 5) the antecedents Γ accumulate path conditions and contain possible preconditions. The updates $\mathcal{U}$ record the current symbolic value at each point during program execution and the ϕ's represent postconditions.

Symbolic execution of a program `p` works as follows:

1. Select an open proof goal with a $[\cdot]$ modality. If no $[\cdot]$ exists on any branch, then symbolic execution is completed. Focus on the first active statement (possibly empty) of the program in the modality.
2. If it is a complex statement, apply rules to decompose it into simple statements and goto 1., otherwise continue.
3. Apply the sequent calculus rule corresponding to the active statement.
4. Simplify the resulting updates and apply first-order simplification to the premises. This might result in some closed branches. It is possible to detect and eliminate infeasible paths in this way. Goto 1.

Example 3. We look at typical proof goals that arise during symbolic execution:

1. Γ, `i > j` $\Rightarrow \mathcal{U}$[`if (i>j) {p} else {q}` $\omega]\phi$: Applying rule ifElse and simplification eliminates the `else` branch and symb. exec. continues with `p` ω.
2. $\Gamma \Rightarrow$ {`i :=c`$\| \ldots\}$[`j = i;` $\omega]\phi$ where `c` is a constant: It is sound to replace the statement `j = i` with `j = c` and continue with symbolic execution. This is known as *constant propagation*. More techniques for *partial evaluation* can be integrated into symbolic execution [12].
3. $\Gamma \Rightarrow$ {`o1.a := v1`$\| \ldots\}$[`o2.a = v2;` $\omega]\phi$: After executing `o2.a = v2`, the *alias* is analyzed as follows: (i) if `o2 = null` is true the program does not terminate; (ii) else, if `o2 = o1` holds, the value of `o1.a` in the update is overriden and the new update is {`o1.a := v2`$\| \ldots \|$`o2.a := v2`}; (iii) else the new update is {`o1.a := v1`$\| \ldots \|$`o2.a := v2`}. Neither of (i)–(iii) might be provable and symbolic execution split into these three cases when encountering a possibly aliased object access.

The result of symbolic execution for a PL program `p` following the sequent calculus rules is a *symbolic execution tree (SET)*, as illustrated in Fig. 6.

Complete symbolic execution trees are finite acyclic trees whose root is labeled with $\Gamma \Longrightarrow [\mathtt{p}]\phi, \Delta$ and no leaf has a $[\cdot]$ modality. Without loss of generality, we can assume that each inner node i is annotated by a sequent $\Gamma_i \Longrightarrow \mathcal{U}_i[\mathtt{p_i}]\phi_i, \Delta_i$, where $\mathtt{p_i}$ is the program to be executed. Every child node is generated by rule application from its parent. A *branching node* represents a statement whose execution causes branching, e.g., conditional, object access, loops etc.

$$\text{emptyBox}\ \frac{\Gamma \Longrightarrow \mathcal{U}\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[]\phi, \Delta}$$

$$\text{assignment}\ \frac{\Gamma \Longrightarrow \mathcal{U}\{\mathtt{x} := litVar\}[\omega]\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\mathtt{x} = litVar; \omega]\phi, \Delta}$$

$$\text{assignAddition}\ \frac{\Gamma \Longrightarrow \mathcal{U}\{\mathtt{x} := litVar_1 + litVar_2\}[\omega]\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\mathtt{x} = litVar_1 + litVar_2; \omega]\phi, \Delta}$$

$$\text{writeAttribute}\ \frac{\Gamma, \mathcal{U}\neg(\mathtt{o} \doteq \mathtt{null}) \Longrightarrow \mathcal{U}\{o.a := se\}[\pi\,\omega]\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[o.a = se;\ \omega]\phi, \Delta}$$

$$\text{ifElse}\ \frac{\Gamma, \mathcal{U}\mathtt{b} \Longrightarrow \mathcal{U}[\mathtt{p}; \omega]\phi, \Delta \qquad \Gamma, \mathcal{U}\neg\mathtt{b} \Longrightarrow \mathcal{U}[\mathtt{q}; \omega]\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\mathtt{if}\ (\mathtt{b})\ \{\mathtt{p}\}\ \mathtt{else}\ \{\mathtt{q}\}\ \omega]\phi, \Delta}$$

$$\text{loopUnwind}\ \frac{\Gamma \Longrightarrow \mathcal{U}[\mathtt{if}\ (exp)\ \{\mathtt{p}; \mathtt{while}\ (exp)\ \{\mathtt{p}\}\}\ \omega]\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\mathtt{while}\ (exp)\ \{\mathtt{p}\}\ \omega]\phi, \Delta}$$

$$\text{loopInvariant}\ \frac{\begin{array}{ll}\Gamma \Longrightarrow \mathcal{U}inv, \Delta & \text{(init)}\\ \Gamma, \mathcal{U}\mathcal{V}_{mod}(\mathtt{b} \wedge inv) \Longrightarrow \mathcal{U}\mathcal{V}_{mod}[\mathtt{p}]inv, \Delta & \text{(preserves)}\\ \Gamma, \mathcal{U}\mathcal{V}_{mod}(\neg\mathtt{b} \wedge inv) \Longrightarrow \mathcal{U}\mathcal{V}_{mod}[\omega]\phi, \Delta & \text{(use case)}\end{array}}{\Gamma \Longrightarrow \mathcal{U}[\mathtt{while}\ (\mathtt{b})\ \{\mathtt{p}\}\ \omega]\phi, \Delta}$$

$$\text{methodInvocation}\ \frac{\begin{array}{l}\Gamma, \mathcal{U}\neg(\mathtt{o} \doteq \mathtt{null}) \Longrightarrow \{\mathcal{U}\}[\\ \qquad \mathtt{if}\ (\mathtt{o}\ \mathtt{instanceof}\ T_n)\ \mathtt{res} = \mathtt{o.m(se)}@T_n;\\ \qquad \mathtt{else\ if}(\mathtt{o}\ \mathtt{instanceof}\ T_{n-1})\ \mathtt{res} = \mathtt{o.m(se)}@T_{n-1};\\ \qquad \ldots\\ \qquad \mathtt{else\ res} = \mathtt{o.m(se)}@T_1;\\ \qquad \omega]\phi, \Delta\end{array}}{\Gamma \Longrightarrow \mathcal{U}[\mathtt{res} = \mathtt{o.m(se)};\ \omega]\phi, \Delta}$$

$$\text{methodContract}\ \frac{\begin{array}{l}\Gamma \Longrightarrow \mathcal{U}\{param_1 := \mathtt{v_1} \| \ldots \| param_n := \mathtt{v_n}\}pre, \Delta\\ \Gamma \Longrightarrow \mathcal{U}\{param_1 := \mathtt{v_1} \| \ldots \| param_n := \mathtt{v_n}\}\mathcal{V}_{mod}(post \rightarrow [\mathtt{r} = res; \omega]\phi), \Delta\end{array}}{\Gamma \Longrightarrow \mathcal{U}[\mathtt{r} = \mathtt{m}(\mathtt{v_1}, \ldots, \mathtt{v_n});\ \omega]\phi, \Delta}$$

For decomposition of complex expressions:

$$\text{postInc}\ \frac{\Gamma \Longrightarrow \mathcal{U}[\mathtt{T}_y\ \mathtt{v_1} = y; y = y + 1; \mathtt{x} = \mathtt{v_1}; \omega]\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\mathtt{x} = y{+}{+}; \omega]\phi, \Delta}$$

$$\text{assignAdditionUnfold}\ \frac{\Gamma \Longrightarrow \mathcal{U}[\mathtt{T}_{exp_1}\ \mathtt{v_1} = exp_1; \mathtt{T}_{exp_2}\ \mathtt{v_2} = exp_2;\ \mathtt{x} = \mathtt{v_1} + \mathtt{v_2}; \omega]\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\mathtt{x} = exp_1 + exp_2; \omega]\phi, \Delta}$$

$$\text{writeAttributeUnfold}\ \frac{\Gamma \Longrightarrow \mathcal{U}[T_{nse}\ \mathtt{v_1} = nse; \mathtt{v_1}.a = se;\ \omega]\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[nse.a = se;\ \omega]\phi, \Delta}$$

$$\text{ifElseUnfold}\ \frac{\Gamma \Longrightarrow \mathcal{U}[\mathtt{boolean\ b}\ =\ nse;\ \mathtt{if}\ (\mathtt{b})\ \{\mathtt{p}\}\ \mathtt{else}\ \{\mathtt{q}\}\ \omega]\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\mathtt{if}\ (nse)\ \{\mathtt{p}\}\ \mathtt{else}\ \{\mathtt{q}\}\ \omega]\phi, \Delta}$$

Fig. 5. Selected sequent calculus rules (for more detail see [3]).

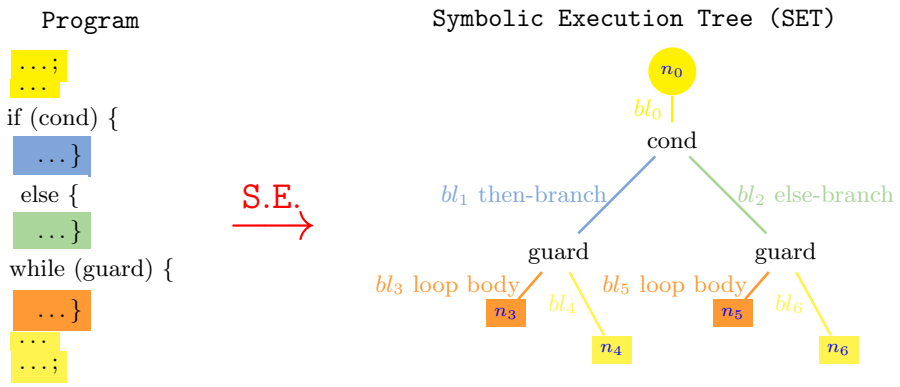

Fig. 6. Symbolic execution tree with loop invariant applied.

Definition 7 (Sequential block). *A* sequential block *(*SB*) is a maximal program fragment in an SET that is symbolically executed without branching.*

For instance, there are 7 sequential blocks bl_0,...,bl_6 in the SET in Fig. 6.

Definition 8 (Child, descendent and sibling sequential block). *For sequential blocks bl_0 and bl_1:*

- *bl_1 is the* child *of bl_0, if bl_0 ends in a branching node n and bl_1 starts with n.*
- *bl_1 is the* descendant *of bl_0, if there exists sequential blocks bl^0,...,bl^m,$0 < m$ such that bl_0=bl^0, bl_1=bl^m and each bl^{i+1} is the child of bl^i for $0 \leq i < m$. Intuitively when $m = 1$, a child is also a descendant.*
- *bl_1 is the* sibling *of bl_0, if both bl_0 and bl_1 starts with the same branching node n .*

In the SET in Fig. 6, bl_3 is the child of bl_0, the sibling of bl_4 and the descendant of bl_0.

Definition 9 (Generalized sequential block). *A* generalized sequential block *(*GSB*) is a sequential block together with all its* descendant *sequential blocks.*

It is a recursive definition, so a GSB always ends with leaf nodes. In the SET in Fig. 6, we have GSB $\{bl_1, bl_3, bl_4\}$ and $\{bl_2, bl_5, bl_6\}$. However, $\{bl_0, bl_1, bl_2, bl_5, bl_6\}$ is not a GSB because bl_1 does not end with leaf nodes. Another remark is that a program is a GSB itself, which is $\{bl_0, bl_1, bl_2, bl_3, bl_4, bl_5, bl_6\}$ in this SET. For convenience, we refer to a GSB with the father sequential block. For instance, GSB $\{bl_1, bl_3, bl_4\}$ is denoted as GSB(bl_1).

4 Sequent Calculus for Program Transformation

The structure of a symbolic execution tree makes it possible to synthesize a program by bottom-up traversal. The idea is to apply the sequent calculus rules reversely and generate the program step-by-step. This requires to extend the sequent calculus rules with means for program synthesis. Obviously, the synthesized program should behave exactly as the original one, at least for the *observable locations*. To this end we introduce the notion of *weak bisimulation* for PL programs and show its soundness for program transformation.

4.1 Weak Bisimulation Relation of Program

Definition 10 (Location sets, observation equivalence). *A* location set *is a set containing program variables* $\mathtt{x}$ *and attribute expressions* $o.\mathtt{a}$ *with* $\mathtt{a} \in \mathsf{Attr}$ *and* o *being a term of the appropriate sort.*

Given two states s_1, s_2 *and a location set obs,* $obs \subseteq loc$*. A relation* $\approx: loc \times S \times S$ *is an* observation equivalence *if and only if for all* $ol \in obs$*,* $val_{D,s_1,\beta}(ol) = val_{D,s_2,\beta}(ol)$ *holds. It is written as* $s_1 \approx_{obs} s_2$*. We call obs* observable locations*.*

The semantics of a PL program $\mathtt{p}$ (Fig. 3) is a state transformation. Executing $\mathtt{p}$ from a start state s results in a set of end states S', where S' is a singleton $\{s'\}$ if $\mathtt{p}$ terminates, or $\emptyset$ otherwise. We identify a singleton with its only member, so in case of termination, $val_{D,s}(\mathtt{p})$ is evaluated to s' instead of $\{s'\}$.

A *transition relation* $\longrightarrow: \Pi \times S \times S$ relates two states s, s' by a program $\mathtt{p}$ iff $\mathtt{p}$ starts in state s and terminates in state s', written $s \xrightarrow{\mathtt{p}} s'$. We have: $s \xrightarrow{\mathtt{p}} s'$, where $s' = val_{D,s}(\mathtt{p})$. If $\mathtt{p}$ does not terminate, we write $s \xrightarrow{\mathtt{p}}$.

Since a complex statement can be decomposed into a set of simple statements, which is done during symbolic execution, we can assume that a program $\mathtt{p}$ consists of simple statements. Execution of $\mathtt{p}$ leads to a sequence of state transitions: $s \xrightarrow{\mathtt{p}} s' \equiv s_0 \xrightarrow{\mathtt{sSt}_0} s_1 \xrightarrow{\mathtt{sSt}_1} \ldots \xrightarrow{\mathtt{sSt}_{n-1}} s_n \xrightarrow{\mathtt{sSt}_n} s_{n+1}$, where $s = s_0$, $s' = s_{n+1}$, s_i a *program state* and $\mathtt{sSt}_\mathtt{i}$ a simple statement $(0 \leq i \leq n)$. A program state has the same semantics as the *state* defined in a Kripke structure, so we use both notations without distinction.

Some simple statements reassign values (write) to a location ol in the observable locations that affects the evaluation of ol in the final state. We distinguish these simple statements from those that do not affect the observable locations.

Definition 11 (Observable and internal statement/transition). *Consider states* s, s'*, a simple statement* $\mathtt{sSt}$*, a transition relation* $\longrightarrow$*, where* $s \xrightarrow{\mathtt{sSt}} s'$*, and the observable locations obs; we call* $\mathtt{sSt}$ *an* observable statement *and* $\longrightarrow$ *an* observable transition*, if and only if there exists* $ol \in obs$*, and* $val_{D,s',\beta}(ol) \neq val_{D,s,\beta}(ol)$*. We write* $\xrightarrow{\mathtt{sSt}}_{obs}$*. Otherwise,* $\mathtt{sSt}$ *is called an* internal statement *and* $\longrightarrow$ *an* internal transition*, written* $\longrightarrow_{int}$*.*

In this definition, observable/internal transitions are *minimal* transitions that relate two states with a simple statement. We indicate the simple statement $\mathtt{sSt}$

in the notion of the observable transition $\xrightarrow{\texttt{sSt}}_{obs}$, since `sSt` reflects the changes of the observable locations. In contrast, an internal statement does not appear in the notion of the internal transition.

Example 4. Given the set of observable locations *obs*={`x`, `y`}, the simple statement "`x = 1 + z;`" is observable, because `x`'s value is reassigned. The statement "`z = x + y;`" is internal, since the evaluation of `x`, `y` are not changed, even though the value of each variable is read by `z`.

Remark. An observable transition is defined by observing the changes of *obs* in the *final* state after the transition. For a program that consists of many statements, the observable locations for the final state may differ from that for some internal state. Assume an observable transition $s \xrightarrow{\texttt{sSt}}_{obs} s'$ changes the evaluation of some location $ol \in obs$ in state s'. The set of observable locations obs_1 in state s should also contain the locations ol_1 that is *read* by ol, because the change to ol_1 can lead to a change of ol in the final state s'.

Example 5. Consider the set of observable locations *obs*={`x`, `y`} and program fragment "`z = x + y; x = 1 + z;`". The statement `z = x + y;` becomes observable because the value of `z` is changed and it will be used later in the observable statement `x = 1 + z;`. The observable location set obs_1 should contain `z` after the execution of `z = x + y;` .

Definition 12 (Weak transition). *Given a set of observable locations obs, the transition relation $\Longrightarrow_{int}$ is the reflexive and transitive closure of $\longrightarrow_{int}$: $s \Longrightarrow_{int} s'$ holds iff for states $s_0,\ldots,s_n$, $n \geq 0$, we have $s = s_0$, $s' = s_n$ and $s_0 \longrightarrow_{int} s_1 \longrightarrow_{int} \cdots \longrightarrow_{int} s_n$. In the case of $n = 0$, $s \Longrightarrow_{int} s$ holds. The transition relation $\stackrel{\texttt{sSt}}{\Longrightarrow}_{obs}$ is the composition of the relations $\Longrightarrow_{int}$, $\xrightarrow{\texttt{sSt}}_{obs}$ and $\Longrightarrow_{int}$: $s \stackrel{\texttt{sSt}}{\Longrightarrow}_{obs} s'$ holds iff there are states s_1 and s_2 such that $s \Longrightarrow_{int} s_1 \xrightarrow{\texttt{sSt}}_{obs} s_2 \Longrightarrow_{int} s'$. The* weak transition $\stackrel{\widehat{\texttt{sSt}}}{\Longrightarrow}_{obs}$ *represents either $\stackrel{\texttt{sSt}}{\Longrightarrow}_{obs}$, if* `sSt` *observable or $\Longrightarrow_{int}$ otherwise.*

In other words, a weak transition is a sequence of minimal transitions that contains at most one observable transition.

Definition 13 (Weak bisimulation for states). *Given two programs $\texttt{p}_1, \texttt{p}_2$ and observable locations obs, obs′, let $\texttt{sSt}_1$ be a simple statement and s_1, s_1' two program states of $\texttt{p}_1$, and $\texttt{sSt}_2$ is a simple statement and s_2, s_2' are two program states of $\texttt{p}_2$. A relation $\approx$ is a* weak bisimulation *for states if and only if $s_1 \approx_{obs} s_2$ implies:*

- *if $s_1 \stackrel{\widehat{\texttt{sSt}_1}}{\Longrightarrow}_{obs'} s_1'$, then $s_2 \stackrel{\widehat{\texttt{sSt}_2}}{\Longrightarrow}_{obs'} s_2'$ and $s_1' \approx_{obs'} s_2'$*
- *if $s_2 \stackrel{\widehat{\texttt{sSt}_2}}{\Longrightarrow}_{obs'} s_2'$, then $s_1 \stackrel{\widehat{\texttt{sSt}_1}}{\Longrightarrow}_{obs'} s_1'$ and $s_2' \approx_{obs'} s_1'$*

where $val_{D,s_1}(\texttt{sSt}_1) \approx_{obs'} val_{D,s_2}(\texttt{sSt}_2)$.

Definition 14 (Weak bisimulation for programs). *Let* $\mathtt{p}_1, \mathtt{p}_2$ *be two programs, obs and obs′ are observable locations, and* $\approx$ *is a weak bisimulation relation for states.* $\approx$ *is a* weak bisimulation *for programs, written* $\mathtt{p}_1 \approx_{obs} \mathtt{p}_2$*, if for the sequence of state transitions:*

$s_1 \xrightarrow{\mathtt{p}_1} s_1' \equiv s_1^0 \xrightarrow{\mathtt{sSt}_1^0} s_1^1 \xrightarrow{\mathtt{sSt}_1^1} \ldots \xrightarrow{\mathtt{sSt}_1^{n-1}} s_1^n \xrightarrow{\mathtt{sSt}_1^n} s_1^{n+1}$*, with* $s_1 = s_1^0$*,* $s_1' = s_1^{n+1}$*,*

$s_2 \xrightarrow{\mathtt{p}_2} s_2' \equiv s_2^0 \xrightarrow{\mathtt{sSt}_2^0} s_2^1 \xrightarrow{\mathtt{sSt}_2^1} \ldots \xrightarrow{\mathtt{sSt}_2^{m-1}} s_1^m \xrightarrow{\mathtt{sSt}_2^m} s_2^{m+1}$*, with* $s_2 = s_2^0$*,* $s_2' = s_2^{m+1}$*,*

we have (i) $s_2' \approx_{obs} s_1'$*; (ii) for each state* s_1^i *there exists a state* s_2^j *such that* $s_1^i \approx_{obs'} s_2^j$ *for some obs′; (iii) for each state* s_2^j *there exists a state* s_1^i *such that* $s_2^j \approx_{obs'} s_1^i$ *for some obs′, where* $0 \leq i \leq n$ *and* $0 \leq j \leq m$*.*

The weak bisimulation relation for programs defined above requires a weak transition that relates two states with at most one observable transition. This definition reflects the *structural* properties of a program and can be characterized as a *small-step semantics* [13]. It directly implies the lemma below that relates the weak bisimulation relation of programs to a *big-step semantics* [14].

Lemma 1. *Let* $\mathtt{p}, \mathtt{q}$ *be programs and obs the set of observable locations. It holds* $\mathtt{p} \approx_{obs} \mathtt{q}$ *if and only if for any first-order structure D and state s,* $val_{D,s}(\mathtt{p}) \approx_{obs} val_{D,s}(\mathtt{q})$ *holds.*

4.2 The Weak Bisimulation Modality

We introduce a weak bisimulation modality which allows us to relate two programs that behave indistinguishably on the observable locations.

Definition 15 (Weak bisimulation modality—syntax). *The* bisimulation modality $[\ \mathtt{p}\ \langle\!\rangle\ \mathtt{q}\ ]@(obs, use)$ *is a modal operator providing compartments for programs* $\mathtt{p}$*,* $\mathtt{q}$ *and location sets obs and use. We extend our definition of formulas: Let* ϕ *be a* PL*-DL formula and* $\mathtt{p}, \mathtt{q}$ *two* PL *programs and obs, use two location sets such that* $pv(\phi) \subseteq obs$ *where* $pv(\phi)$ *is the set of all program variables occurring in* ϕ*, then* $[\ \mathtt{p}\ \langle\!\rangle\ \mathtt{q}\ ]@(obs, use)\phi$ *is also a* PL*-DL formula.*

The intuition behind the location set $usedVar(s, \mathtt{p}, obs)$ defined below is to capture precisely those locations whose value influences the final value of an observable location $l \in obs$ (or the evaluation of a formula ϕ) after executing a program $\mathtt{p}$. We approximate the set later by the set of all program variables in a program that are used before being redefined (i.e., assigned a new value).

Definition 16 (Used program variable). *A variable* $v \in \mathsf{PV}$ *is called* used by a program $\mathtt{p}$ *with respect to a location set obs, if there exists an* $l \in obs$ *such that*

$$D, s \models \forall v_l.\exists v_0.((\langle \mathtt{p} \rangle l = v_l) \rightarrow (\{v := v_0\}\langle \mathtt{p} \rangle l \neq v_l))$$

The set $usedVar(s, \mathtt{p}, obs)$ *is defined as the smallest set containing all used program variables of* $\mathtt{p}$ *with respect to obs.*

The formula defining a used variable v of a program $\mathtt{p}$ encodes that there is an interference with a location contained in *obs*. In Example 5, $\mathtt{z}$ is a used variable. We formalize the semantics of the weak bisimulation modality:

Definition 17 (Weak bisimulation modality—semantics). *With* $\mathtt{p}, \mathtt{q}$ PL-*programs,* D, s, β*, and obs, use as above, let* $val_{D,s,\beta}([\ \mathtt{p}\ \langle\!\rangle\ \mathtt{q}\ ]@(obs, use)\phi) = tt$ *if and only if*

1. $val_{D,s,\beta}([\mathtt{p}]\phi) = tt$
2. $use \supseteq usedVar(s, \mathtt{q}, obs)$
3. *for all* $s' \approx_{use} s$ *we have* $val_{D,s}(\mathtt{p}) \approx_{obs} val_{D,s'}(\mathtt{q})$

Lemma 2. *Let obs be the set of all locations observable by* ϕ *and let* $\mathtt{p}, \mathtt{q}$ *be programs. If* $\mathtt{p} \approx_{obs} \mathtt{q}$ *then* $val_{D,s,\beta}([\mathtt{p}]\phi) \leftrightarrow val_{D,s,\beta}([\mathtt{q}]\phi)$ *holds for all D, s,* β*.*

Proof. Direct consequence of Definition 17 and Lemma 1. □

An extended sequent for the bisimulation modality is:

$$\Gamma \Longrightarrow \mathcal{U}[\ p\ \langle\!\rangle\ q\ ]@(obs, use)\phi, \Delta$$

The following lemma gives an explicit meaning of used variable set *use*.

Lemma 3. *An extended sequent* $\Gamma \Longrightarrow \mathcal{U}[\ p\ \langle\!\rangle\ q\ ]@(obs, use)\phi, \Delta$ *within a sequential block bl (see Definition 7) represents a certain state* s_1*, where P is the original program of bl, p is the original program to be executed in bl at state* s_1*, and* p' *is the original program already been executed in bl; while Q is program to be generated of bl, q is the already generated program in bl, and* q' *is the remaining program to be generated in bl. The location set use are the* dynamic observable locations *that the following relations hold: (i)* $p \approx_{obs} q$*; (ii)* $P \approx_{obs} Q$*; (iii)* $p' \approx_{use} q'$*.*

Proof. The structure of this sequential block *bl* is illustrated in Fig. 7.

(i) $p \approx_{obs} q$

It is the direct consequence of Definition 17.

(ii) $P \approx_{obs} Q$

Consider the initial state s_0 of this sequential block, where $use = use_0$, p=P and q=Q in the sequent, we have $s'_0 \approx_{use_0} s_0$, according to Definition 17 and Lemma 1, $P \approx_{obs} Q$ holds.

(iii) $p' \approx_{use} q'$

Consider the truncated sequential block bl_2 starting from the current state s_1 and ending with the final state s_2 According to Definition 16, if there is no program in bl_2, then we have $obs = use$. Now consider the truncated sequential block bl_1 starting from the initial state s_0 and ending with the current state s_1. We have $use = use_0$, p=p', q=q' and $obs = use$ in the sequent, according to Definition 17 and Lemma 1, $p' \approx_{use} q'$ holds.

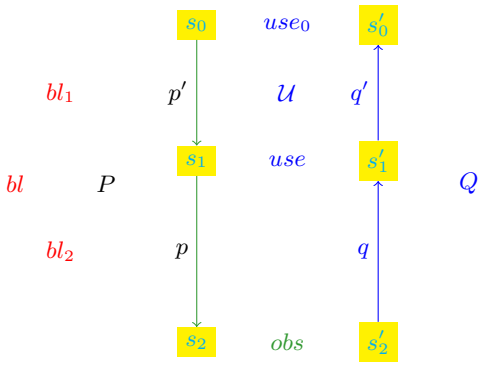

Fig. 7. Program in a sequential block.

4.3 Sequent Calculus Rules for the Bisimulation Modality

The sequent calculus rules for the bisimulation modality are of the following form:

$$\mathsf{ruleName}\ \frac{\begin{array}{c}\Gamma_1 \Longrightarrow \mathcal{U}_1[\ p_1\ \langle\!\rangle\ q_1\ ]@(obs_1, use_1)\phi_1, \Delta_1 \\ \cdots \\ \Gamma_n \Longrightarrow \mathcal{U}_n[\ p_n\ \langle\!\rangle\ q_n\ ]@(obs_n, use_n)\phi_n, \Delta_n\end{array}}{\Gamma \Longrightarrow \mathcal{U}[\ p\ \langle\!\rangle\ q\ ]@(obs, use)\phi, \Delta}$$

Fig. 8 shows some extended sequent calculus rules, where $\overline{\omega}$ denotes the generated program that is weak bisimilar to ω. Unlike standard sequent calculus rules that are executed from root to leaves, sequent rule application for the bisimulation modality consists of two phases:

Phase 1. Symbolic execution of source program p as usual. In addition, the observable location sets obs_i are propagated, since they contain the locations observable by $\mathtt{p_i}$ and ϕ_i that will be used in the second phase. Typically, *obs* contains the return variables of a method and the locations used in the continuation of the program, e.g., program variables used after a loop must be reflected in the observable locations of the loop body. The result of this phase is a symbolic execution tree as illustrated in Fig. 6.

Phase 2. We synthesize the target program `q` and used variable set `use` from $\mathtt{q}_i$ and $\mathtt{use_i}$ by applying the rules in a leave-to-root manner. One starts with a leaf node and generates the program within its sequential block first, e.g., bl_3, bl_4, bl_5, bl_6 in Fig. 6. These are combined by rules corresponding to statements that contain a sequential block, such as loopInvariant (containing bl_3 and bl_4). One continues with the generalized sequential block containing the compound statements, e.g., $GSB(bl_2)$, and so on, until the root is reached. Note that the order of processing the sequential blocks matters, for instance, the program for the sequential block bl_4 must be generated before that for bl_3, because the observable locations in node n_3 depend on the used variable set of bl_4 according to the loopInvariant rule.

Now we show the program transformation in action.

$$\textsf{emptyBox}\ \frac{\Gamma \Longrightarrow \mathcal{U}@(obs, \_)\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}[\ \texttt{nop}\ \langle\!\rangle\ \texttt{nop}\ ]@(obs, obs)\phi, \Delta}$$

$$\textsf{assignment}\ \frac{\Gamma \Longrightarrow \mathcal{U}\{\texttt{l} := \texttt{r}\}[\ \omega\ \langle\!\rangle\ \overline{\omega}\ ]@(obs, use)\phi, \Delta}{\begin{pmatrix} \Gamma \Longrightarrow \mathcal{U}[\ l = r; \omega\ \langle\!\rangle\ l = r; \overline{\omega}\ ]@(obs, use - \{l\} \cup \{r\})\phi, \Delta & \text{if } l \in use \\ \Gamma \Longrightarrow \mathcal{U}[\ l = r; \omega\ \langle\!\rangle\ \overline{\omega}\ ]@(obs, use)\phi, \Delta & otherwise \end{pmatrix}}$$

$$\textsf{ifElse}\ \frac{\begin{array}{c} \Gamma, \mathcal{U}b \Longrightarrow \mathcal{U}[\ p; \omega\ \langle\!\rangle\ \overline{p;\omega}\ ]@(obs, use_{p;\omega})\phi, \Delta \\ \Gamma, \mathcal{U}\neg b \Longrightarrow \mathcal{U}[\ q; \omega\ \langle\!\rangle\ \overline{q;\omega}\ ]@(obs, use_{q;\omega})\phi, \Delta \end{array}}{\begin{array}{l} \Gamma \Longrightarrow \mathcal{U}[\ \texttt{if}\ (b)\ \{p\}\ \texttt{else}\ \{q\}; \omega\ \langle\!\rangle \\ \qquad \texttt{if}\ (b)\ \{\overline{\texttt{p};\omega}\}\ \texttt{else}\ \{\overline{\texttt{q};\omega}\}\ ]@(obs, use_{p;\omega} \cup use_{q;\omega} \cup \{b\})\phi, \Delta \end{array}}$$

(with b boolean variable.)

$$\textsf{loopUnwind}\ \frac{\begin{array}{l} \Gamma \Longrightarrow \mathcal{U}[\ \texttt{if}\ (\texttt{b})\ \{\texttt{p}; \texttt{while}\ (\texttt{b})\ \{\texttt{p}\}\}\ \omega\ \langle\!\rangle \\ \qquad \overline{\texttt{if}\ (\texttt{b})\ \{\texttt{p}; \texttt{while}\ (\texttt{b})\ \{\texttt{p}\}\}\ \omega}\ ]@(obs, use)\phi, \Delta \end{array}}{\Gamma \Longrightarrow \mathcal{U}[\ \texttt{while(b)}\ \{\texttt{p}\}\ \omega\ \langle\!\rangle\ \overline{\texttt{if}\ (\texttt{b})\ \{\texttt{p}; \texttt{while(b)}\ \{\texttt{p}\}\}\ \omega}\ ]@(obs, use)\phi, \Delta}$$

$$\textsf{loopInvariant}\ \frac{\begin{array}{l} \Gamma \Longrightarrow \mathcal{U}inv, \Delta \\ \Gamma, \mathcal{U}\mathcal{V}_{mod}(\texttt{b} \wedge inv) \Longrightarrow \mathcal{U}\mathcal{V}_{mod}[\ p\ \langle\!\rangle\ \overline{p}\ ]@(use_1 \cup \{b\}, use_2)inv, \Delta \\ \Gamma, \mathcal{U}\mathcal{V}_{mod}(\neg\texttt{b} \wedge inv) \Longrightarrow \mathcal{U}\mathcal{V}_{mod}[\ \omega\ \langle\!\rangle\ \overline{\omega}\ ]@(obs, use_1)\phi, \Delta \end{array}}{\Gamma \Longrightarrow \mathcal{U}[\ \texttt{while(b)}\{\texttt{p}\}\,\omega\ \langle\!\rangle\ \texttt{while(b)}\{\overline{\texttt{p}}\}\,\overline{\omega}\ ]@(obs, use_1 \cup use_2 \cup \{b\})\phi, \Delta}$$

$$\textsf{methodContract}_{\texttt{C}=(pre,post,mod)}\ \frac{\begin{array}{l} \Gamma \Longrightarrow \mathcal{U}\{prm_1 := \texttt{v}_1 \| \ldots \| prm_n := \texttt{v}_n\}pre, \Delta \\ \Gamma \Longrightarrow \mathcal{U}\{prm_1 := \texttt{v}_1 \| \ldots \| prm_n := \texttt{v}_n\}\mathcal{V}_{mod} \\ \qquad (post \rightarrow \{\texttt{r} := res\}[\ \omega\ \langle\!\rangle\ \overline{\omega}\ ]@(obs, use)\phi), \Delta \end{array}}{\begin{array}{l} \Gamma \Longrightarrow \mathcal{U}[\ \texttt{r} = \texttt{m}(\texttt{v}_1, \ldots, \texttt{v}_n);\ \omega\ \langle\!\rangle\ \texttt{r} = \texttt{m}(\texttt{v}_1, \ldots, \texttt{v}_n); \overline{\omega}\ ]@(obs, use)\phi, \Delta \\ \quad \text{(Contract } C \text{ is correct)} \end{array}}$$

Fig. 8. A collection of sequent calculus rules for program transformation.

Example 6. Given observable locations obs={x}, we perform program transformation for the following PL program.

```
y = y + z;
if (b) {
  y = z++;
  x = z;
}
else {
  z = 1;
  x = y + z;
  y = x;
  x = y + 2;
}
```

In the first phase, we do symbolic execution using the extended sequent calculus shown in Fig. 8. We use sp_i to denote the program to be generated, and use_i to denote the used variable set. To ease the presentation, we omit postcondition ϕ, as well as unnecessary formulas Γ and Δ. The first active statement is an assignment, so the assignment rule is applied. A conditional is encountered. After the application of ifElse rule, the result is the symbolic execution tree shown in Fig. 9.

$$\frac{\mathcal{U}_1\mathtt{b} \Longrightarrow \mathcal{U}_1[\ y = z + +; \ldots \ \langle\!\rangle\ sp_2\ ]@(\{\mathtt{x}\}, use_2) \quad \mathcal{U}_1\neg\mathtt{b} \Longrightarrow \mathcal{U}_1[\ z = 1; \ldots \ \langle\!\rangle\ sp_3\ ]@(\{\mathtt{x}\}, use_3)}{\frac{\Longrightarrow \{\mathtt{y} := \mathtt{y} + \mathtt{z}\}[\ \mathtt{if(b)\{\ldots\}else\{\ldots\}}\ \langle\!\rangle\ sp_1\ ]@(\{\mathtt{x}\}, use_1)}{\Longrightarrow [\ \mathtt{y} = \mathtt{y} + \mathtt{z}; \ldots \ \langle\!\rangle\ sp_0\ ]@(\{\mathtt{x}\}, use_0)}}$$

Fig. 9. Symbolic execution tree until conditional.

Now the symbolic execution tree splits into 2 branches. $\mathcal{U}_1$ denotes the update computed in the previous steps: $\{\mathtt{y} := \mathtt{y} + \mathtt{z}\}$. We first concentrate on the `then`-branch, where the condition `b` is `TRUE`. The first active statement $\mathtt{y} = \mathtt{z} + +;$ is a complex statement. We decompose it into 3 simple statements using the postInc rule introduced in Fig. 5. Then after a few applications of the assignment rule followed by the emptyBox rule, the symbolic execution tree in this sequential block is shown in Fig. 10.

Now the source program is empty, so we can start generating a program for this sequential block. By applying the emptyBox rule in the other direction, we get sp_8 as `nop` and use_8={x}. The next rule application is assignment. Because $x \in use_8$, the assignment $x = z;$ is generated and the used variable set is updated by removing x but adding z. So we have sp_7: $x = z;$ and use_7={z}. In the next step, despite another assignment rule application, no statement is generated because $y \notin use_7$, and sp_6 and use_6 are identical to sp_7 and use_7. Following 3 more assignment rule applications, in the end we get sp_2: $z = z + 1; x = z;$ and

$$\mathcal{U}_1\mathtt{b} \Longrightarrow \mathcal{U}_1\{\mathtt{t} := \mathtt{z}\}\{\mathtt{z} := \mathtt{z}+1\}\{\mathtt{y} := \mathtt{t}\}\{\mathtt{x} := \mathtt{z}\}@(\{\mathtt{x}\}, \_)$$

$$\mathcal{U}_1\mathtt{b} \Longrightarrow \mathcal{U}_1\{\mathtt{t} := \mathtt{z}\}\{\mathtt{z} := \mathtt{z}+1\}\{\mathtt{y} := \mathtt{t}\}\{\mathtt{x} := \mathtt{z}\}[\ \lozenge\ sp_8\ ]@(\{\mathtt{x}\}, use_8)$$

$$\mathcal{U}_1\mathtt{b} \Longrightarrow \mathcal{U}_1\{\mathtt{t} := \mathtt{z}\}\{\mathtt{z} := \mathtt{z}+1\}\{\mathtt{y} := \mathtt{t}\}[\ x = z;\ \lozenge\ sp_7\ ]@(\{\mathtt{x}\}, use_7)$$

$$\mathcal{U}_1\mathtt{b} \Longrightarrow \mathcal{U}_1\{\mathtt{t} := \mathtt{z}\}\{\mathtt{z} := \mathtt{z}+1\}[\ y = t; \ldots\ \lozenge\ sp_6\ ]@(\{\mathtt{x}\}, use_6)$$

$$\mathcal{U}_1\mathtt{b} \Longrightarrow \mathcal{U}_1\{\mathtt{t} := \mathtt{z}\}[\ z = z+1;\ y = t; \ldots\ \lozenge\ sp_5\ ]@(\{\mathtt{x}\}, use_5)$$

$$\mathcal{U}_1\mathtt{b} \Longrightarrow \mathcal{U}_1[\ int\ t = z;\ z = z+1;\ y = t; \ldots\ \lozenge\ sp_4\ ]@(\{\mathtt{x}\}, use_4)$$

$$\mathcal{U}_1\mathtt{b} \Longrightarrow \mathcal{U}_1[\ y = z++; \ldots\ \lozenge\ sp_2\ ]@(\{\mathtt{x}\}, use_2)$$

Fig. 10. Symbolic execution tree of then branch.

use_2=$\{z\}$. So $z = z + 1; x = z;$ is the program synthesized in this sequential block.

So far we have done the program transformation for the `then`-branch. Analogous to this, we can generate the program for the `else`-branch. After the first phase of symbolic execution, the symbolic execution tree is built as shown in Fig. 11. In the second phase, the program is synthesized after applying a sequence of assignment rules. The resulting program for this sequential block is sp_3: $z = 1; x = y + z; y = x; x = y + 2;$, while use_3=$\{y\}$.

$$\mathcal{U}_1\neg\mathtt{b} \Longrightarrow \mathcal{U}_1\{\mathtt{z} := \mathtt{1}\}\{\mathtt{x} := \mathtt{y}+\mathtt{z}\}\{\mathtt{y} := \mathtt{x}\}\{\mathtt{x} := \mathtt{y}+\mathtt{2}\}@(\{\mathtt{x}\}, \_)$$

$$\mathcal{U}_1\neg\mathtt{b} \Longrightarrow \mathcal{U}_1\{\mathtt{z} := \mathtt{1}\}\{\mathtt{x} := \mathtt{y}+\mathtt{z}\}\{\mathtt{y} := \mathtt{x}\}\{\mathtt{x} := \mathtt{y}+\mathtt{2}\}[\ \lozenge\ sp_{12}\ ]@(\{\mathtt{x}\}, use_{12})$$

$$\mathcal{U}_1\neg\mathtt{b} \Longrightarrow \mathcal{U}_1\{\mathtt{z} := \mathtt{1}\}\{\mathtt{x} := \mathtt{y}+\mathtt{z}\}\{\mathtt{y} := \mathtt{x}\}[\ x = y + 2;\ \lozenge\ sp_{11}\ ]@(\{\mathtt{x}\}, use_{11})$$

$$\mathcal{U}_1\neg\mathtt{b} \Longrightarrow \mathcal{U}_1\{\mathtt{z} := \mathtt{1}\}\{\mathtt{x} := \mathtt{y}+\mathtt{z}\}[\ y = x; \ldots\ \lozenge\ sp_{10}\ ]@(\{\mathtt{x}\}, use_{10})$$

$$\mathcal{U}_1\neg\mathtt{b} \Longrightarrow \mathcal{U}_1\{\mathtt{z} := \mathtt{1}\}[\ x = y + z; \ldots\ \lozenge\ sp_9\ ]@(\{\mathtt{x}\}, use_9)$$

$$\mathcal{U}_1\neg\mathtt{b} \Longrightarrow \mathcal{U}_1[\ z = 1; \ldots\ \lozenge\ sp_3\ ]@(\{\mathtt{x}\}, use_3)$$

Fig. 11. Symbolic execution tree of else branch.

Now we have synthesized the program for both sequential blocks. Back to the symbolic execution tree shown in Fig. 9, we can build a conditional by applying the ifElse rule. The result is sp_1: $if(b)\ \{z = z + 1; x = z;\}\ else\ \{z = 1; x = y + z; y = x; x = y + 2;\}$, and use_1=$\{b, z, y\}$. After a final assignment rule application, the program generated is shown in Fig. 12.

Remark. Our approach to program transformation will generate a program that only consists of simple statements. The generated program is optimized to a certain degree, because the used variable set avoids generating unnecessary statements. In this sense, our program transformation framework can be considered as *program specialization*. In fact, during the symbolic execution phase, we can interleave partial evaluation actions, i.e., constant propagation, deadcode-

```
y = y + z;
if (b) {
  z = z + 1;
  x = z;
}
else {
  z = 1;
  x = y + z;
  y = x;
  x = y + 2;
}
```

Fig. 12. The generated program for Example 6.

elimination, safe field access and type inference ([12]). It will result in a more optimized program.

5 Soundness

Theorem 1. *The extended sequent calculus rules are sound.*

The deductive description of the presented program transformation rule system enables us to reuse standard proof techniques applied in soundness proofs for classical logic calculi.

The basic approach is to prove soundness for each rule. The soundness of the whole method is then a consequence of the soundness theorem for classical sequent calculi $\vdash$:

Theorem 2. *If all rules of the proof system $\vdash$ are sound, then the proof system is sound.*

The soundness proof for the classical calculus rules remains unchanged. The interesting part is the soundness proof for the rules dealing with the weak bisimulation modality. The soundness proof of these rules requires in particular to show, that the transformed program is equivalent to the original one up to weak bisimulation with respect to a specified set of observable locations *obs*.

We need first some lemmas which establish simple properties that are mostly direct consequences of the respective definitions given in the Section 4.2.

The following lemma allows us to extend the weak bisimulation relation for two states when we know that they coincide on the value of x.

Lemma 4. *Let $s_1, s_2 \in S$ be observation equivalent $s_1 \approx_{obs} s_2$ and $x : T \in \mathsf{PV}$. If $s_1(x) = s_2(x)$ then $s_1 \approx_{obs \cup \{x\}} s_2$.*

Proof. Direct consequence of Definition 10. □

The next lemma states that two bisimilar states remain bisimular if both are updated by identical assignments:

Lemma 5. *Let $s_1, s_2 \in S$ be observation equivalent $s_1 \approx_{obs} s_2$. If s_1', s_2' are such that $s_1' = s_1[x \leftarrow d]$ and $s_2' = s_2[x \leftarrow d]$ for a program variable $x : T$ and domain element $d \in D(T)$ then $s_1' \approx_{obs} s_2'$.*

Proof. Direct consequence of Definition 10. □

We need further that the bisimulation relation is anti-monotone with respect to the set of observable locations.

Lemma 6. *Given two programs $\mathtt{p}, \mathtt{q}$ and location sets loc_1, loc_2 with $loc_1 \subseteq loc_2$. If $\mathtt{p} \approx_{loc_2} \mathtt{q}$ then also $\mathtt{p} \approx_{loc_1} \mathtt{q}$.*

Proof. Direct consequence of Definition 14. □

Finally, we need the fact that changes to unobserved locations have no effect on the bisimulation relation between two states:

Lemma 7. *Let loc denote a set of locations, $l : T \in \mathsf{PV}$ and $s_1, s_2 \in S$.*
If $l \notin loc$ and $s_1 \approx_{loc} s_2$ then for all $d \in \mathcal{D}_T$:

$$s_1[l \leftarrow d] \approx_{loc} s_2$$

Proof. Direct consequence of Definition 10. □

We can now turn to the soundness proof for the calculus rules. We prove here exemplarily that the assignment rule for local variables is sound. The rule is central to the approach as it performs a state change.

Lemma 8. *The rule*

assignment

$$\frac{\Gamma \Longrightarrow \mathcal{U}\{\mathtt{l} := \mathtt{r}\}[\,\omega \between \overline{\omega}\,]@(obs, use)\phi, \Delta}{\begin{pmatrix} \Gamma \Longrightarrow \mathcal{U}[\,l = r; \omega \between l = r; \overline{\omega}\,]@(obs, use - \{l\} \cup \{r\})\phi, \Delta & \text{if } l \in use \\ \Gamma \Longrightarrow \mathcal{U}[\,l = r; \omega \between \overline{\omega}\,]@(obs, use)\phi, \Delta & otherwise \end{pmatrix}}$$

with l, r local variables

is sound.

Proof. To check the soundness of the rule, we have to prove that if all premises of the rule are valid then its conclusion is also valid.

We fix a first-order structure D, a state s and a variable assignment β. Further, we assume that for all formulas $\gamma \in \Gamma$: $val_{D,s,\beta}(\gamma) = tt$ and for all formulas $\delta \in \Delta$: $val_{D,s,\beta}(\Delta) = ff$ holds. Otherwise, the conclusion is trivially satisfied by D, s, β. Hence, we can assume that

$$val_{D,s,\beta}(\mathcal{U}\{l := r\}[\,\omega \between \overline{\omega}\,]@(obs, use)\phi) = tt$$

or, equivalently,

$$val_{D,\widehat{s},\beta}([\ \omega\ \between\ \overline{\omega}\ ]@(obs, use)\phi) = tt \tag{1}$$

where

$$s_{\mathcal{U}} := val_{D,s,\beta}(\mathcal{U})(s),\quad \widehat{s} := val_{D,s_{\mathcal{U}},\beta}(l := r)(s_{\mathcal{U}}) = val_{D,s,\beta}(\mathcal{U} \| \mathcal{U}(l := r))(s)$$

holds.

Case 1 ($l \in use$):

We have to show that

$$\begin{aligned} & val_{D,s,\beta}(\mathcal{U}[\ l = r; \omega\ \between\ l = r; \overline{\omega}\ ]@(obs, use')\phi) \\ = {} & val_{D,s_{\mathcal{U}},\beta}([\ l = r; \omega\ \between\ l = r; \overline{\omega}\ ]@(obs, use')\phi) \\ = {} & tt \end{aligned}$$

with $use' := use - \{l\} \cup \{r\}$ holds.

To prove that $val_{D,s_{\mathcal{U}},\beta}([\ l = r; \omega\ \between\ l = r; \overline{\omega}\ ]@(obs, use')\phi) = tt$ we need to check the three items of Definition 17:

Item 1 is satisfied if

$$val_{D,s,\beta}(\mathcal{U}[l = r; \omega]\phi) = tt$$

holds. This is a direct consequence from the correctness of the sequent calculus presented in Section 4.3.

Item 2 $use' \supseteq usedVar(s, l = r; \overline{\omega}, obs)$ expresses that use' captures at least all used variables and it is a direct consequence of the definition of $usedVar$. By assumption use contains at least all variables actually read by $\overline{\omega}$. The program $l = r; \overline{\omega}$ redefines l which can be safely removed from use while variable r is read and needs to be added.

Item 3 is the last remaining item that needs to be proven, i.e., that the two programs in the conclusion are actually weak bisimular with respect to the location set obs.

We have to show that for all $s_1 \approx_{use'} s_{\mathcal{U}}$:

$$val_{D,s_{\mathcal{U}}}(l = r; \omega)\ \approx_{obs}\ val_{D,s_1}(l = r; \overline{\omega})$$

holds. Following the semantics definitions given in Fig. 3 we get

$$val_{D,s_{\mathcal{U}}}(l = r; \omega) = \bigcup_{s' \in val_{D,s_{\mathcal{U}}}(l=r;)} val_{D,s'}(\omega) = val_{D,\widehat{s}}(\omega)$$

and

$$val_{D,s_1}(l = r; \overline{\omega}) = \bigcup_{s'_1 \in val_{D,s_1}(l=r;)} val_{D,s'_1}(\overline{\omega}) = val_{D,\widehat{s_1}}(\overline{\omega})\ \text{ with } \{\widehat{s_1}\} = val_{D,s_1}(l = r;)$$

As use' contains r and because $s_1 \approx_{use'} s_{\mathcal{U}}$ we get

$$s_{\mathcal{U}}(r) = s_1(r) \tag{2}$$

and, hence,

$$\widehat{s}(l) = \widehat{s_1}(l) \tag{3}$$

Applying Lemma 5 we get

$$\begin{aligned} & \widehat{s} \approx_{use'} \widehat{s_1} \\ \Leftrightarrow\ & \widehat{s} \approx_{use-\{l\}\cup\{r\}} \widehat{s_1} \\ \overset{\Rightarrow}{\text{Lemma 6}}\ & \widehat{s} \approx_{use-\{l\}} \widehat{s_1} \\ \overset{\Rightarrow}{(3)}\ & \widehat{s} \approx_{use} \widehat{s_1} \end{aligned}$$

With assumption (1) and Definition 15, we get $val_{D,\widehat{s}}(\omega) \approx_{obs} val_{D,\widehat{s_1}}(\overline{\omega})$ and hence

$$val_{D,s_{\mathcal{U}}}(l = r;\omega) = val_{D,\widehat{s}}(\omega) \approx_{obs} val_{D,\widehat{s_1}}(\overline{\omega}) = val_{D,s_1}(l = r;\overline{\omega})$$

Case 2 ($l \notin use$): As for case 1 we have to check all three items. The first item is identical to case 1 and the second item is trivial as the transformed program does not change. Item 3 remains to be checked, i.e., for an arbitrary s_1 with

$$s_1 \approx_{use'} s_{\mathcal{U}} \tag{4}$$

we have to prove that

$$val_{D,s_{\mathcal{U}}}(l = r;\omega) \ \approx_{obs}\ val_{D,s_1}(\overline{\omega})$$

holds (i.e., that the final states are observation equivalent), we have to use the fact that $l \notin use$ and that item 2 holds, i.e., that use contains at least all variables read by $\overline{\omega}$.

$$\begin{aligned} & s_1 \approx_{use'} s_{\mathcal{U}} \\ \Rightarrow\ & s_1 \approx_{use} s_{\mathcal{U}} \\ \overset{\Rightarrow}{Lemma\ 7}\ & s_1 \approx_{use} \widehat{s} \\ \overset{\Rightarrow}{(1)}\ & val_{D,\widehat{s}}(\omega) \ \approx_{obs}\ val_{D,s_1}(\overline{\omega}) \\ \overset{\Rightarrow}{(1)}\ & val_{D,s_{\mathcal{U}}}(l = r;\omega) = val_{D,\widehat{s}}(\omega) \ \approx_{obs}\ val_{D,s_1}(\overline{\omega}) \end{aligned}$$

□

We conclude this section with a short discussion of the loop invariant rule. The interesting aspect of the loop invariant rule is that the observable location set *obs* of the second premise differs from the others. This allows us to establish a connection to the notion of a program context as used in compositional correctness proofs.

Compositional compiler correctness proofs consider the context $C(\circ)$ in which the compiled entity p is *used*. A context C is a description contain the placeholder $\circ$ which can be instantiated by 'any' program entity q.

The idea is to formalize a stable interface on which p can rely on and with which p interacts. A compositional compiler must now be able to compile p such that a given correctness criteria are satisfied for the compilation $\mathtt{p}_{compiled}$ with respect to C.

The observable location set *obs* in the presented approach is similar to the context as described above. It specifies which effects must be preserved by the compiler (program transformer). E.g., when the program p to be transformed is a method body, then the observable set contains only the location which refers to the result value of the method and implicitly, all heap locations.

If the effect on these locations produced by the transformed program is indistinguishable from the respective effect of the original program, then the program transformer is considered correct. In case of the loop invariant rule, the loop body is transformed *independently* in the second branch. It would not be enough to just use the original context instead, we must demand that all effects on local variables used by the code following the loop statement as well as the loop guard variable are preserved.

6 Optimization

The previously introduced program transformation technique generates a program that consists only of simple statements. With the help of the used variable set, we avoid generating unnecessary statements, so the program is optimized to a certain level. An optimization can be made to interleave partial evaluation actions with symbolic execution in the first phase.

6.1 Sequentialized Normal Form of Updates

Updates reflect the state of program execution. In particular, the update in a sequential block records the evaluation of the locations in that sequential block. We can involve updates in the second phase of program generation, which leads to further optimization opportunities. As defined in Definition 5, updates in normal form are in the form of single static assignment (SSA). It is easy to maintain normal form of updates in a sequential block when applying the extended sequent calculus rules of Fig. 8. This can be used for further optimization of the generated program.

Take the assignment rule for example: after each forward rule application, we do an update simplification step to maintain the normal form of the update for that sequential block; when a statement is synthesized by applying the rule backwards, we use the *update* instead of the executed assignment statement, to obtain the value of the location to be assigned; then we generate the assignment statement with that value.

Example 7. Consider the following program:

```
i = j + 1;
j = i;
i = j + 1;
```

After executing the first two statements and update simplification, we obtain the normal form update $\mathcal{U}_2^{nf} = \{\mathtt{i} := \mathtt{j}+1 \| \mathtt{j} := \mathtt{j}+1\}$. Doing the same with the third statement results in $\mathcal{U}_3^{nf} = \{\mathtt{j} := \mathtt{j}+1 \| \mathtt{i} := \mathtt{j}+2\}$, which implies that in the final state `i` has value $\mathtt{j}+2$ and `j` has value $\mathtt{j}+1$.

Let `i` be the only observable location, for which a program is now synthesized bottom-up, starting with the third statement. The rules in Fig. 8 would allow to generate the statement `i = j + 1;`. But, reading the value of location `i` from $\mathcal{U}_3^{nf}$ as sketched above, the statement `i = j + 2;` is generated. This reflects the current value of `j` along the sequential block and saves an assignment.

A first attempt to formalize our ideas is the following assignment rule:

$$\frac{\Gamma \Longrightarrow \mathcal{U}_1^{nf}[\ \omega\ \between\ \overline{\omega}\ ]@(obs, use)\phi, \Delta}{\begin{pmatrix} \Gamma \Longrightarrow \mathcal{U}^{nf}[\ l = r; \omega\ \between\ l = r_1; \overline{\omega}\ ]@(obs, use - \{l\} \cup \{r\})\phi, \Delta & \text{if } l \in use \\ \Gamma \Longrightarrow \mathcal{U}^{nf}[\ l = r; \omega\ \between\ \overline{\omega}\ ]@(obs, use)\phi, \Delta & otherwise \end{pmatrix}}$$

with $\mathcal{U}_1^{nf} = \{\ldots \| \mathtt{l} := \mathtt{r_1}\}$ being the normal form of $\mathcal{U}^{nf}\{\mathtt{l} := \mathtt{r}\}$

However, this rule is not sound. If we continue Example 7 with synthesizing the first two assignments, we obtain `j = j + 1; i = j + 2;` by using the new rule, which is clearly incorrect, because `i` has final value $\mathtt{j}+3$ instead of $\mathtt{j}+2$. The problem is that the values of locations in the normal form update are independently synthesized from each other and do not reflect how one statement is affected by the execution of previous statements in sequential execution. To ensure correct usage of updates in program generation, we introduce the concept of a *sequentialized normal form* (SNF) of an update. Intuitively, it is the update of the normal form in which every involved assignment statement is independent of each other.

Definition 18 (Elementary update independence). *An elementary update* $l_1 := exp_1$ *is* independent *from another elementary update* $l_2 := exp_2$*, if* l_1 *does not occur in* exp_2 *and* l_2 *does not occur in* exp_1*.*

Definition 19 (Sequentialized normal form update). *An update is in* sequentialized normal form*, denoted by* $\mathcal{U}^{snf}$*, if it has the shape of a sequence of two parallel updates* $\{u_1^a \| \ldots \| u_m^a\}\{u_1 \| \ldots \| u_n\}$*,* $m \geq 0, n \geq 0$*.*

$\{u_1 \| \ldots \| u_n\}$ *is the* core *update, denoted by* $\mathcal{U}^{snf_c}$*, where each* u_i *is an* elementary update *of the form* $l_i := exp_i$*, and all* u_i*,* u_j *(*$i \neq j$*) are independent and have no conflict.*

$\{u_1^a \| \ldots \| u_m^a\}$ *is the* auxiliary *update, denoted by* $\mathcal{U}^{snf_a}$*, where (i) each* u_i^a *is of the form* $l^k := l$ *(*$k \geq 0$*); (ii)* l *is a program variable; (iii)* l^k *is a* fresh *program variable not occurring anywhere else in* $\mathcal{U}^{snf_a}$ *and not occurring in the location set of the core update* $l^k \notin \{l_i | 0 \leq i \leq n\}$*; (iv) there is no conflict between* u_i^a *and* u_j^a *for all* $i \neq j$*.*

Any normal form update whose elementary updates are independent is also an SNF update that has only a core part.

Example 8 (SNF update). For the following updates,

- $\{i^0 := i \| i^1 := i\}\{i := i^0 + 1 \| j := i^1\}$ is in sequentialized normal form.
- $\{i^0 := j \| i^1 := i\}\{i := i^0 + 1 \| j := i^1\}$ and $\{i^0 := i + 1 \| i^1 := i\}\{i := i^0 + 1 \| j := i^1\}$ are not in sequentialized normal form: $i^0 := j$ has different base variables on the left and right, while $i^0 := i + 1$ has a complex term on the right, both contradicting (i).
- $\{i^0 := i \| i^1 := i\}\{i := i^0 + 1 \| j := i\}$ is not in sequentialized normal form, because $i := i^0 + 1$ and $j := i$ are not independent.

To compute the SNF of an update, in addition to the rules given in Fig. 4 we need two more rules shown in Fig. 13.

(*associativity*) $\{u_1\}\{u_2\}\{u_3\} \rightsquigarrow \{u_1\}(\{u_2\}\{u_3\})$
(*introducing auxiliary*) $\{u\} \rightsquigarrow \{\mathtt{x}^0 := \mathtt{x}\}(\{\mathtt{x} := \mathtt{x}^0\}\{u\})$, where $\mathtt{x}^0 \notin pv$

Fig. 13. Rules for computing SNF updates.

Lemma 9. *The associativity rule and introducing auxiliary rule are sound.*

Proof. We use the update simplification rules defined in Fig. 4 to prove these two rules.

Associativity

The left hand side:

$$\begin{aligned} &\{u_1\}\{u_2\}\{u_3\} \\ \rightsquigarrow\ &\{u_1 \| \{u_1\}u_2\}\{u_3\} \\ \rightsquigarrow\ &\{u_1 \| \{u_1\}u_2 \| \{u_1 \| \{u_1\}u_2\}u_3\} \end{aligned}$$

The right hand side:

$$\begin{aligned} &\{u_1\}(\{u_2\}\{u_3\}) \\ \rightsquigarrow\ &\{u_1\}\{u_2 \| \{u_2\}u_3\} \\ \rightsquigarrow\ &\{u_1 \| \{u_1\}(u_2 \| \{u_2\}u_3)\} \\ \rightsquigarrow\ &\{u_1 \| \{u_1\}u_2 \| \{u_1\}\{u_2\}u_3\} \\ \rightsquigarrow\ &\{u_1 \| \{u_1\}u_2 \| \{u_1 \| \{u_1\}u_2\}u_3\} \end{aligned}$$

So, $\{u_1\}\{u_2\}\{u_3\} = \{u_1\}(\{u_2\}\{u_3\})$. We have proved the associativity rule.

Introducing auxiliary

The right hand side:

$$
\begin{aligned}
&\{\mathtt{x}^0 := \mathtt{x}\}(\{\mathtt{x} := \mathtt{x}^0\}\{u\}) \\
\rightsquigarrow\ &\{\mathtt{x}^0 := \mathtt{x}\}\{\mathtt{x} := \mathtt{x}^0\}\{u\} \text{ (associativity)} \\
\rightsquigarrow\ &\{\mathtt{x}^0 := \mathtt{x} \| \{\mathtt{x}^0 := \mathtt{x}\}\mathtt{x} := \mathtt{x}^0\}\{u\} \\
\rightsquigarrow\ &\{\mathtt{x}^0 := \mathtt{x} \| \mathtt{x} := \mathtt{x}\}\{u\} \\
\rightsquigarrow\ &\{\mathtt{x} := \mathtt{x}\}\{u\} \text{ (since } \mathtt{x}^0 \notin pv) \\
\rightsquigarrow\ &\{u\}
\end{aligned}
$$

So the introducing auxiliary rule is proven. □

We can maintain the SNF of an update on a sequential block as follows: after executing a program statement, apply the associativity rule and compute the core update; if the newly added elementary update $l := r$ is not independent from some update in the core, then apply introducing auxiliary rule to introduce $\{l^0 := l\}$, then compute the new auxiliary update and core update.

6.2 Sequent Calculus Rules Involving Updates

With the help of the SNF of an update, a sound assignment rule can be given as follows:

assignment

$$\frac{\Gamma \Longrightarrow \mathcal{U}_1^{snf}[\,\omega \between \overline{\omega}\,]@(obs, use)\phi, \Delta}{\begin{pmatrix} \Gamma \Longrightarrow \mathcal{U}^{snf}[\,l = r; \omega \between l = r_1; \overline{\omega}\,]@(obs, use - \{l\} \cup \{r\})\phi, \Delta & \text{if } l \in use \\ \Gamma \Longrightarrow \mathcal{U}^{snf}[\,l = r; \omega \between \overline{\omega}\,]@(obs, use)\phi, \Delta & otherwise \end{pmatrix}}$$

where $\mathcal{U}_1^{snf} = \mathcal{U}_1^{snf_a}\{\ldots \| \mathtt{l} := \mathtt{r}_1\}$ is the SNF of $\mathcal{U}^{snf}\{\mathtt{l} := \mathtt{r}\}$).

Whenever the core update is empty, the following auxAssignment rule is used, which means the auxiliary assignments are always generated in the beginning of a sequential block.

auxAssignment

$$\frac{\Gamma \Longrightarrow \mathcal{U}_1^{snf_a}[\,\omega \between \overline{\omega}\,]@(obs, use)\phi, \Delta}{\begin{pmatrix} \Gamma \Longrightarrow \mathcal{U}^{snf_a}[\,\omega \between \mathtt{T}_l\ l^0 = l; \overline{\omega}\,]@(obs, use - \{l^0\} \cup \{l\})\phi, \Delta & \text{if } l^0 \in use \\ \Gamma \Longrightarrow \mathcal{U}^{snf_a}[\,\omega \between \overline{\omega}\,]@(obs, use)\phi, \Delta & otherwise \end{pmatrix}}$$

where $\mathcal{U}^{snf_a} = \{u\}$ and $\mathcal{U}_1^{snf_a} = \{u \| \mathtt{l}^0 := \mathtt{l}\}$ being the auxiliary update

Most of the other rules are obtained by replacing $\mathcal{U}$ with $\mathcal{U}^{snf}$. Some are shown in Fig. 14.

Example 9. We demonstrate that the program from Example 7 is now handled correctly. After executing the first two statements and simplifying the update, we get the normal form update $\mathcal{U}_2^{nf} = \{\mathtt{i} := \mathtt{j}{+}1 \| \mathtt{j} := \mathtt{j}{+}1\}$. Here a dependency issue occurs, so we introduce the auxiliary update $\{\mathtt{j}^0 := \mathtt{j}\}$ and simplify to the sequentialized normal form update $\mathcal{U}_2^{snf} = \{\mathtt{j}^0 := \mathtt{j}\}\{\mathtt{i} := \mathtt{j}^0 + 1 \| \mathtt{j} := \mathtt{j}^0 + 1\}$. Continuing with the third statement and performing update simplification results in the SNF update $\mathcal{U}_3^{snf} = \{\mathtt{j}^0 := \mathtt{j}\}\{\mathtt{j} := \mathtt{j}^0 + 1 \| \mathtt{i} := \mathtt{j}^0 + 2\}$. By applying the rules above, we synthesize the program `int` $\mathtt{j}^0 = \mathtt{j}; \mathtt{i} = \mathtt{j}^0{+}2;$, which still saves one assignment and is sound.

$$\textsf{emptyBox}\ \frac{\Gamma \Longrightarrow \mathcal{U}^{snf}@(obs, \_)\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}^{snf}[\ \texttt{nop}\ \wr\ \texttt{nop}\ ]@(obs, obs)\phi, \Delta}$$

assignment

$$\frac{\Gamma \Longrightarrow \mathcal{U}_1^{snf}[\ \omega\ \wr\ \overline{\omega}\ ]@(obs, use)\phi, \Delta}{\begin{pmatrix} \Gamma \Longrightarrow \mathcal{U}^{snf}[\ l = r; \omega\ \wr\ l = r_1; \overline{\omega}\ ]@(obs, use - \{l\} \cup \{r\})\phi, \Delta & \text{if } l \in use \\ \Gamma \Longrightarrow \mathcal{U}^{snf}[\ l = r; \omega\ \wr\ \overline{\omega}\ ]@(obs, use)\phi, \Delta & otherwise \end{pmatrix}}$$

(with $\mathcal{U}_1^{snf} = \mathcal{U}_1^{snf_a}\{\ldots \| \texttt{l} := \texttt{r}_1\}$ the SNF of $\mathcal{U}^{snf}\{\texttt{l} := \texttt{r}\}$)

auxAssignment

$$\frac{\Gamma \Longrightarrow \mathcal{U}_1^{snf_a}[\ \omega\ \wr\ \overline{\omega}\ ]@(obs, use)\phi, \Delta}{\begin{pmatrix} \Gamma \Longrightarrow \mathcal{U}^{snf_a}[\ \omega\ \wr\ l^0 = l; \overline{\omega}\ ]@(obs, use - \{l^0\} \cup \{l\})\phi, \Delta & \text{if } l^0 \in use \\ \Gamma \Longrightarrow \mathcal{U}^{snf_a}[\ \omega\ \wr\ \overline{\omega}\ ]@(obs, use)\phi, \Delta & otherwise \end{pmatrix}}$$

(with $\mathcal{U}^{snf_a} = \{u\}$ and $\mathcal{U}_1^{snf_a} = \{u \| \texttt{l}^0 := \texttt{l}\}$ being the auxiliary update)

$$\textsf{ifElse}\ \frac{\begin{array}{l}\Gamma, \mathcal{U}^{snf} b \Longrightarrow \mathcal{U}^{snf}[\ p; \omega\ \wr\ \overline{p; \omega}\ ]@(obs, use_{p;\omega})\phi, \Delta \\ \Gamma, \mathcal{U}^{snf} \neg b \Longrightarrow \mathcal{U}^{snf}[\ q; \omega\ \wr\ \overline{q; \omega}\ ]@(obs, use_{q;\omega})\phi, \Delta\end{array}}{\begin{array}{l}\Gamma \Longrightarrow \mathcal{U}^{snf}[\ \texttt{if}\ (b)\ \{p\}\ \texttt{else}\ \{q\}; \omega\ \wr \\ \qquad \texttt{if}\ (b)\ \{\overline{\texttt{p;}\omega}\}\ \texttt{else}\ \{\overline{\texttt{q;}\omega}\}\ ]@(obs, use_{p;\omega} \cup use_{q;\omega} \cup \{b\})\phi, \Delta\end{array}}$$

(with b boolean variable.)

$$\textsf{loopUnwind}\ \frac{\Gamma \Longrightarrow \mathcal{U}^{snf}[\ \texttt{if (b) \{p; while (b) \{p\}\}}\ \omega\ \wr\ \overline{\texttt{if (b) \{p; while (b) \{p\}\}}\ \omega}\ ]@(obs, use)\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}^{snf}[\ \texttt{while(b) \{p\}}\ \omega\ \wr\ \overline{\texttt{if (b) \{p; while(b) \{p\}\}}\ \omega}\ ]@(obs, use)\phi, \Delta}$$

$$\textsf{loopInvariant}\ \frac{\begin{array}{l}\Gamma \Longrightarrow \mathcal{U}^{snf} inv, \Delta \\ \Gamma, \mathcal{U}^{snf}\mathcal{V}_{mod}(\texttt{b} = \texttt{TRUE} \wedge inv) \Longrightarrow \mathcal{U}^{snf}\mathcal{V}_{mod} \\ \qquad\qquad [\ p\ \wr\ \overline{p}\ ]@(use_1 \cup \{b\}, use_2) inv, \Delta \\ \Gamma, \mathcal{U}^{snf}\mathcal{V}_{mod}(\texttt{b} = \texttt{FALSE} \wedge inv) \Longrightarrow \mathcal{U}^{snf}\mathcal{V}_{mod}[\ \omega\ \wr\ \overline{\omega}\ ]@(obs, use_1)\phi, \Delta\end{array}}{\Gamma \Longrightarrow \mathcal{U}^{snf}[\ \texttt{while(b)\{p\}}\ \omega\ \wr\ \texttt{while(b)\{}\overline{\texttt{p}}\texttt{\}}\ \overline{\omega}\ ]@(obs, use_1 \cup use_2 \cup \{b\})\phi, \Delta}$$

$\textsf{methodContract}_{\texttt{C}=(pre, post, mod)}$

$$\frac{\begin{array}{l}\Gamma \Longrightarrow \mathcal{U}^{snf}\{prm_1 := \texttt{v}_1 \| \ldots \| prm_n := \texttt{v}_n\} pre, \Delta \\ \Gamma \Longrightarrow \mathcal{U}^{snf}\{prm_1 := \texttt{v}_1 \| \ldots \| prm_n := \texttt{v}_n\}\mathcal{V}_{mod} \\ \qquad\qquad (post \rightarrow \{\texttt{r} := res\}[\ \omega\ \wr\ \overline{\omega}\ ]@(obs, use)\phi), \Delta\end{array}}{\begin{array}{l}\Gamma \Longrightarrow \mathcal{U}^{snf}[\ \texttt{r} = \texttt{m}(\texttt{v}_1, \ldots, \texttt{v}_n);\ \omega\ \wr\ \texttt{r} = \texttt{m}(\texttt{v}_1, \ldots, \texttt{v}_n); \overline{\omega}\ ]@(obs, use)\phi, \Delta \\ \quad \text{(Contract } C \text{ is correct)}\end{array}}$$

Fig. 14. A collection of sequent calculus rules for program transformation using SNF update.

Remark. Remember that the program is synthesized within a sequential block first and then constructed. The SNF updates used in the above rules are the SNF updates in the current sequential block. A program execution path may contain several sequential blocks. We do keep the SNF update for each sequential block without simplifying them further into a bigger SNF update for the entire execution path. For example in Fig. 6, the execution path from node n_0 to n_4 involves 3 sequential blocks bl_0, bl_1 and bl_4. When we synthesize the program in bl_4, more precisely, we should write $\mathcal{U}_0^{snf}\mathcal{U}_2^{snf}\mathcal{U}_4^{snf}$ to represent the update used in the rules. However, we just care about the SNF update of bl_4 when generating the program for bl_4, so in the above rules, $\mathcal{U}^{snf}$ refers to $\mathcal{U}_4^{snf}$ and the other SNF updates are omitted.

Theorem 3. *The extended sequent calculus rules involving updates are sound.*

Proof. Follows from the soundness of the extended sequent calculus rules (Theorem 1), the update simplification rules (Fig. 4) and Lemma 9. □

Now we revisit Example 6 and show how to generate a more optimized program.

Example 10. Given observable locations *obs*={x}, specialize the following PL program by the approach involving updates in the program generation phase.

```
y = y + z;
if (b) {
  y = z++;
  x = z;
}
else {
  z = 1;
  x = y + z;
  y = x;
  x = y + 2;
}
```

In the first phase, we do symbolic execution using the extended sequent calculus rules involving updates given in Fig. 14. We ignore the postcondition ϕ and unnecessary formulas Γ and Δ. To ease the presentation, we do not mention the update simplification step all the time, but keep in mind that updates within a sequential block are always simplified after each rule application. Also, we just show the sequents computed after sequent calculus rule application and update simplification, but hide the intermediate ones before simplifying the updates. As usual, sp_i denotes the program to be generated, and use_i denotes the used variable set.

The first active statement is an assignment, we apply the assignment rule. After the application of the ifElse rule, the result is the symbolic execution tree shown in Fig. 15. Here, $\mathcal{U}_1^{snf}$ denotes the sequentialized normal formed update $\{\mathtt{y} := \mathtt{y} + \mathtt{z}\}$. Note that in the path condition, now we only have b (or $\neg$b) instead

$$\frac{\mathtt{b} \Longrightarrow \mathcal{U}_1^{snf}[\ y = z + +; \ldots \ \langle\!\rangle\ sp_2\ ]@(\{\mathtt{x}\}, use_2) \qquad \neg\mathtt{b} \Longrightarrow \mathcal{U}_1^{snf}[\ z = 1; \ldots \ \langle\!\rangle\ sp_3\ ]@(\{\mathtt{x}\}, use_3)}{\Longrightarrow \{\mathtt{y} := \mathtt{y} + \mathtt{z}\}[\ \mathtt{if(b)\{\ldots\}else\{\ldots\}}\ \langle\!\rangle\ sp_1\ ]@(\{\mathtt{x}\}, use_1)}$$
$$\Longrightarrow [\ \mathtt{y} = \mathtt{y} + \mathtt{z}; \ldots \ \langle\!\rangle\ sp_0\ ]@(\{\mathtt{x}\}, use_0)$$

Fig. 15. Symbolic execution tree until conditional.

of $\mathcal{U}_1^{snf}\mathtt{b}$ (or $\mathcal{U}_1^{snf}\neg\mathtt{b}$). It is the result of update simplification after applying the ifElse rule.

Now the symbolic execution tree splits into 2 branches.

We symbolically execute the `then`-branch first. The complex statement $\mathtt{y} = \mathtt{z} + +;$ is decomposed into 3 simple statements using the postInc rule. After the application of the assignment rule on $t = z;$, the resulting update is $\{\mathtt{t} := \mathtt{z}\}$. It is an SNF update that only contains the core part. Then we apply the assignment rule on $z = z + 1;$. The update we get before simplification is $\{\mathtt{t} := \mathtt{z}\}\{\mathtt{z} := \mathtt{z} + 1\}$. To simplify this update, we first transform it into parallel form $\{\mathtt{t} := \mathtt{z} \| \mathtt{z} := \mathtt{z} + 1\}$ using the rules given in Fig. 4. Notice that $\mathtt{z}$, on the left hand side of $\mathtt{z} := \mathtt{z} + 1$, occurs on the right hand side of $\mathtt{t} := \mathtt{z}$, so the elementary updates $\mathtt{t} := \mathtt{z}$ and $\mathtt{z} := \mathtt{z} + 1$ are not independent. To obtain an SNF update, we use the introducing auxiliary rule defined in Fig. 13. So the update is rewritten as $\{\mathtt{z}^0 := \mathtt{z}\}(\{\mathtt{z} := \mathtt{z}^0\}\{\mathtt{t} := \mathtt{z} \| \mathtt{z} := \mathtt{z} + 1\})$, where $\mathtt{z}^0$ is a fresh variable and the auxiliary update $\{\mathtt{z}^0 := \mathtt{z}\}$ is introduced. After simplifying the core part, we finally get the SNF update $\{\mathtt{z}^0 := \mathtt{z}\}\{\mathtt{t} := \mathtt{z}^0 \| \mathtt{z} := \mathtt{z}^0 + 1\}$. From now on, after a few steps application of assignment rule followed by the emptyBox rule, the symbolic execution tree in this sequential block is shown in Fig. 16.

$$\frac{\neg\mathtt{b} \Longrightarrow \mathcal{U}_1^{snf}\{\mathtt{z}^0 := \mathtt{z}\}\{\mathtt{t} := \mathtt{z}^0 \| \mathtt{z} := \mathtt{z}^0 + 1 \| \mathtt{y} := \mathtt{z}^0 \| \mathtt{x} := \mathtt{z}^0 + 1\}@(\{\mathtt{x}\}, \_)}{\mathtt{b} \Longrightarrow \mathcal{U}_1^{snf}\{\mathtt{z}^0 := \mathtt{z}\}\{\mathtt{t} := \mathtt{z}^0 \| \mathtt{z} := \mathtt{z}^0 + 1 \| \mathtt{y} := \mathtt{z}^0 \| \mathtt{x} := \mathtt{z}^0 + 1\}[\ \langle\!\rangle\ sp_8\ ]@(\{\mathtt{x}\}, use_8)}$$
$$\frac{\mathtt{b} \Longrightarrow \mathcal{U}_1^{snf}\{\mathtt{z}^0 := \mathtt{z}\}\{\mathtt{t} := \mathtt{z}^0 \| \mathtt{z} := \mathtt{z}^0 + 1 \| \mathtt{y} := \mathtt{z}^0\}[\ x = z; \ \langle\!\rangle\ sp_7\ ]@(\{\mathtt{x}\}, use_7)}{\mathtt{b} \Longrightarrow \mathcal{U}_1^{snf}\{\mathtt{z}^0 := \mathtt{z}\}\{\mathtt{t} := \mathtt{z}^0 \| \mathtt{z} := \mathtt{z}^0 + 1\}[\ y = t; \ldots \ \langle\!\rangle\ sp_6\ ]@(\{\mathtt{x}\}, use_6)}$$
$$\frac{\mathtt{b} \Longrightarrow \mathcal{U}_1^{snf}\{\mathtt{t} := \mathtt{z}\}[\ z = z + 1;\ y = t; \ldots \ \langle\!\rangle\ sp_5\ ]@(\{\mathtt{x}\}, use_5)}{\mathtt{b} \Longrightarrow \mathcal{U}_1^{snf}[\ int\ t = z;\ z = z + 1;\ y = t; \ldots \ \langle\!\rangle\ sp_4\ ]@(\{\mathtt{x}\}, use_4)}$$
$$\mathtt{b} \Longrightarrow \mathcal{U}_1^{snf}[\ y = z + +; \ldots \ \langle\!\rangle\ sp_2\ ]@(\{\mathtt{x}\}, use_2)$$

Fig. 16. Symbolic execution tree of then branch.

Now we start generating the program for this sequential block. By applying the emptyBox rule in the other direction, we get sp_8 as `nop` and use_8=$\{\mathtt{x}\}$. In the next step, since $x \in use_8$, the assignment $x = z^0 + 1;$ is generated according to the assignment rule involving SNF update. The used variable set is updated

by removing x but adding z^0. So we have sp_7: $x = z^0 + 1;$ and use_7=$\{z^0\}$. The application of 4 more assignment rules generates no more new statement. Now the core update is empty and we can generate the auxiliary assignment according to the auxAssignment rule. In the end, we get for this sequential branch $sp2 : int\ z^0 = z; x = z^0 + 1;$ and use_2=$\{z\}$.

Analogous to this, we can generate the program for the `else`-branch. After the first phase of symbolic execution while maintaining the SNF update, Fig. 17 shows the resulting symbolic execution tree. In the second phase, the program is synthesized after applying a sequence of assignment rules and a final auxAssignment rule. The result program for this sequential block is $int\ y^0 = y; x = y^0 + 2;$, and use_3=$\{y\}$.

$$\neg \mathtt{b} \Longrightarrow \mathcal{U}_1^{snf}\{\mathtt{y}^0 := \mathtt{y}\}\{\mathtt{z} := 1 \| \mathtt{y} := \mathtt{y}^0 + 1 \| \mathtt{x} := \mathtt{y}^0 + 3\}@(\{\mathtt{x}\}, \_)$$

$$\neg \mathtt{b} \Longrightarrow \mathcal{U}_1^{snf}\{\mathtt{y}^0 := \mathtt{y}\}\{\mathtt{z} := 1 \| \mathtt{y} := \mathtt{y}^0 + 1 \| \mathtt{x} := \mathtt{y}^0 + 3\}[\ \langle\!\rangle\ sp_{12}\ ]@(\{\mathtt{x}\}, use_{12})$$

$$\neg \mathtt{b} \Longrightarrow \mathcal{U}_1^{snf}\{\mathtt{y}^0 := \mathtt{y}\}\{\mathtt{z} := 1 \| \mathtt{x} := \mathtt{y}^0 + 1 \| \mathtt{y} := \mathtt{y}^0 + 1\}[\ x = y + 2;\ \langle\!\rangle\ sp_{11}\ ]@(\{\mathtt{x}\}, use_{11})$$

$$\neg \mathtt{b} \Longrightarrow \mathcal{U}_1^{snf}\{\mathtt{z} := 1 \| \mathtt{x} := \mathtt{y} + 1\}[\ y = x; \ldots\ \langle\!\rangle\ sp_{10}\ ]@(\{\mathtt{x}\}, use_{10})$$

$$\neg \mathtt{b} \Longrightarrow \mathcal{U}_1^{snf}\{\mathtt{z} := 1\}[\ x = y + z; \ldots\ \langle\!\rangle\ sp_9\ ]@(\{\mathtt{x}\}, use_9)$$

$$\neg \mathtt{b} \Longrightarrow \mathcal{U}_1^{snf}[\ z = 1; \ldots\ \langle\!\rangle\ sp_3\ ]@(\{\mathtt{x}\}, use_3)$$

Fig. 17. Symbolic execution tree of else branch.

Now the programs for both sequential blocks are synthesized. We can generate the whole program by applying the ifElse rule and assignment rule. The specialized program is shown in Fig. 18.

```
y = y + z;
if (b) {
  int z^0 = z;
  x = z^0 + 1;
}
else {
  int y^0 = y;
  x = y^0 + 3;
}
```

Fig. 18. The generated program for Example 10.

Compared to the specialization results from Example 6, we get a more optimized program by involving SNF updates during the generation phase. The specialized program introduces auxiliary variables and is not necessarily con-

taining only simple statements (although there are only simple statements in this example). This is more like a real-world program compared to the programs only containing simple statements.

7 Related Work

JSpec [15] is a state-of-the-art program specializer for Java. It uses an *offline* partial evaluation technique that depends on *binding time analysis*. Our work is based on symbolic execution to derive information on-the-fly, similar to *online* partial evaluation [16], however, we do not generate the program during symbolic execution, but synthesize it in the second phase. In principle, our first phase can obtain as much information as online partial evaluation, and the second phase can generate a more precise optimized program. A major advantage of our approach is that the generated program is guaranteed to be correct. There is work on proving the correctness of a partial evaluator by [17], but they need to encode the correctness properties into a logic programming language.

Verifying Compiler [18] project aims at the development of a compiler that verifies the program during compilation. On contrast, our work might be called *Compiling Verifier*, since the optimized program is generated on the basis of a verification system. Recently, compiler verification became possible [19], however, it aims at verifying a full compiler with fixed rules, which is very expensive, while our approach works at a specific target program and is fully automatic.

The product program technique [20] can be used to verify that two closely related programs preserve behavior, but the programs must be given and loop invariants must be supplied. This has been applied for loop vectorization [21], where specific heuristics do away with the need for invariants and target program is synthesized. The main differences to our work are that we aim at general programs and we use a different synthesis principle.

8 Conclusions and Future Work

We presented a sound framework for program transformation and optimization. It employs symbolic execution, deduction and bisimulation to achieve a precise analysis of variable dependencies and aliasing, and yields an optimized program that has the same behavior as the original program with respect to the observable locations. We presented also an improved and sound approach to obtain a more optimized program by involving updates into the program generation.

The language PL in this paper is a subset of Java, but our technique is valid in general. We intend to extend our approaches to full Java. Observable locations need not be restricted to return variables as in here, but, for example, could be publicly observable variables in an information flow setting. We plan to apply our approaches to language-based security. Finally, the bisimulation modality is not restricted to the same source and target programming language, so we plan to generate Java bytecode from Java source code which will result in a deductive Java compiler that guarantees sound and optimizing compilation.

References

1. Alkassar, E., Hillebrand, M.A., Paul, W.J., Petrova, E.: Automated verification of a small hypervisor. In: VSTTE. Volume 6217 of LNCS. (2010) 40–54
2. Baumann, C., Beckert, B., Blasum, H., Bormer, T.: Lessons learned from microkernel verification – specification is the new bottleneck. In: SSV. Volume 102 of EPTCS. (2012) 18–32
3. Beckert, B., Hähnle, R., Schmitt, P., eds.: Verification of Object-Oriented Software: The KeY Approach. Volume 4334 of LNCS. Springer-Verlag (2007)
4. Bubel, R., Hähnle, R., Ji, R.: Program specialization via a software verification tool. In: FMCO. Volume 6957 of LNCS. (2011)
5. Sangiorgi, D.: Introduction to Bisimulation and Coinduction. (2011)
6. King, J.C.: Symbolic execution and program testing. Communications of the ACM **19**(7) (July 1976) 385–394
7. Ahrendt, W., Baar, T., Beckert, B., Bubel, R., Giese, M., Hähnle, R., Menzel, W., Mostowski, W., Roth, A., Schlager, S., Schmitt, P.H.: The KeY tool: integrating object oriented design and formal verification. SoSyM **4**(1) (2005) 32–54
8. Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. MIT Press (2000)
9. Alpern, B., Wegman, M.N., Zadeck, F.K.: Detecting equality of variables in programs. In: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. POPL '88, New York, NY, USA, ACM (1988) 1–11
10. Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Global value numbers and redundant computations. In: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. POPL '88, New York, NY, USA, ACM (1988) 12–27
11. Rümmer, P.: Sequential, parallel, and quantified updates of first-order structures. In: LPAR. Volume 4246 of LNCS., Springer (2006) 422–436
12. Bubel, R., Hähnle, R., Ji, R.: Interleaving symbolic execution and partial evaluation. In: FMCO. (2009) 125–146
13. Plotkin, G.D.: A structural approach to operational semantics. J. Log. Algebr. Program. **60-61** (2004) 17–139
14. Kahn, G.: Natural semantics. In: STACS. (1987) 22–39
15. Schultz, U.P., Lawall, J.L., Consel, C.: Automatic program specialization for Java. ACM-TPLS **25**(4) (2003) 452–499
16. Ruf, E.S.: Topics in online partial evaluation. PhD thesis, Stanford University, Stanford, CA, USA (1993) UMI Order No. GAX93-26550.
17. Hatcliff, J., Danvy, O.: A computational formalization for partial evaluation. Mathematical Structures in Computer Science **7**(5) (1997) 507–541
18. Hoare, T.: The verifying compiler: A grand challenge for computing research. J. ACM **50** (2003) 63–69
19. Leroy, X.: Formal verification of a realistic compiler. CACM **52**(7) (2009) 107–115
20. Barthe, G., Crespo, J.M., Kunz, C.: Relational verification using product programs. In: FM. Volume 6664 of LNCS. (2011) 200–214
21. Barthe, G., Crespo, J.M., Gulwani, S., Kunz, C., Marron, M.: From relational verification to SIMD loop synthesis. In: PPOPP, ACM (2013) 123–134