Program Transformation Based on Symbolic Execution and Deduction*

Ran Ji, 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 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.

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

R.M. Hierons, M.G. Merayo, and M. Bravetti (Eds.): SEFM 2013, LNCS 8137, pp. 289–304, 2013.

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 Sect. 5.

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. Sect. 2 introduces PL and Sect. 3 defines a program logic for it with semantics and a calculus. These are adapted to the requirements of program transformation in Sect. 4. In Sect. 5 we harvest from our effort and add a non-trivial optimization strategy. We close with related work (Sect. 6) and future work (Sect. 7).

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.

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*. 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* | *locVar*`'='`*spExp*`';'` | *locVar*`'='`*spAtr*`';'`
| *spAtr*`'='`*spExp*`';'`
spLvarDecl ::= *Type* `IDENT';'`
spExp ::= `(`*locVar*`.)`$_{opt}$ *spMthdCall* | *spOpExp* | *litVar*
spMthdCall ::= `mthdName'('`*litVar*$_{opt}$`(',' `*litVar*`)*')'`
spOpExp ::= `!` *litVar* | `-`*litVar* | *litVar* *binOpr* *litVar*
litVar ::= *litval* | *locVar* *litval* ::= $\mathbb{Z}$ | `TRUE` | `FALSE` | `null`
binOpr ::= `<` | `<=` | `>=` | `>` | `==` | `&` | `|` | `*` | `/` | `%` | `+` | `-`
locVar ::= `IDENT` *spAtr* ::= *locVar*`.IDENT`

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 `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 `p` is executed and terminates *then* in all reached final states ϕ holds; the latter means that if `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 `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 $\mathcal{C}$:* $\mathsf{Srt} \supseteq \{\mathtt{int}, \mathtt{boolean}\} \cup \{Cl \mid \text{for all classes } Cl \text{ declared in } \mathcal{C}\}$*; (ii) a* partial subtyping order $\preceq$: $\mathsf{Srt} \times \mathsf{Srt}$ *that models the subtype hierarchy of $\mathcal{C}$ faithfully; (iii) infinite sets of* predicate symbols $\mathsf{Pred} := \{p : T_1 \times \ldots \times T_n \mid T_i \in \mathsf{Srt}, n \in \mathbb{N}\}$ *and* 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* $\alpha(p) = T_1 \times \ldots \times T_n$ *and* $\alpha(f) = T_1 \times \ldots \times T_n \to T$ *the* signature *of the predicate/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*, which do not depend on the current state of program execution;*
- *the* program variables $\mathsf{PV} = \{\mathtt{i}, \mathtt{j}, \ldots\}$*, which are non-rigid constants;*
- *the* attribute *function symbols* 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 names if no ambiguity arises.*

(iv) a set of logical variables $\mathsf{LgV} := \{x : T \mid T \in \mathsf{Srt}\}$.

$\Pi_{\Sigma_{\mathcal{C}}}$ 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 $\mathcal{C}$.

Terms t and formulas ϕ are defined as usual, thus omitted here for brevity. We use *updates* u to describe state changes by means of an explicit substitution. 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, with the same meaning as simple assignments. Elementary updates are composed to *parallel updates* $u_1 \| u_2$ and work like simultaneous assignments. Updates u are defined by the grammar $u ::= \mathtt{i} := t \mid t.\mathtt{a} := t \mid u \parallel u \mid \{u\}u$ (where $\mathtt{a} \in \mathsf{Attr}$) together with the usual well-typedness conditions. Updates applied on terms or formulas, written $\{u\}t$ resp. $\{u\}\phi$, are again terms or formulas. Updates applied on terms or formulas, written $\{u\}t$ resp. $\{u\}\phi$, are again terms or formulas. Terms, formulas and updates are evaluated with respect to a PL-DL Kripke structure:

Definition 2 (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 $\preceq$;* `Null` *is always interpreted as a singleton set and subtype of all class types;*
- $I(f) : \mathcal{D}_{T_1} \times \ldots \times \mathcal{D}_{T_n} \to \mathcal{D}_T$ *for each rigid function symbol* $f : T_1 \times \ldots \times T_n \to T \in \mathsf{Func}_r$*;*
- $I(p) \subseteq \mathcal{D}_{T_1} \times \ldots \times \mathcal{D}_{T_n}$ *for each predicate symbol* $p : T_1 \times \ldots \times\ T_n \in \mathsf{Pred}$*;*

and (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.*

$$val_{D,s,\beta}(\mathtt{x} := t)(s) = s[\mathtt{x} \leftarrow t]$$
$$val_{D,s,\beta}(o.\mathtt{a} := t)(s) = s[(\mathtt{a})(val_{D,s,\beta}(o)) \leftarrow t]$$
$$val_{D,s,\beta}(u_1 \| u_2)(s) = val_{D,s,\beta}(u_2)(val_{D,s,\beta}(u_1)(s))$$
$$val_{D,s,\beta}(\{u_1\}u_2)(s) = val_{D,s',\beta}(u_2)(s'), \text{ where } s' = val_{D,s,\beta}(u_1)(s)$$
$$val_{D,s}(\mathtt{x} = e) = \{s'[\mathtt{x} \leftarrow d] \mid (s', d) \in val_{D,s}(e)\},\ \mathtt{x} \in \mathsf{PV}$$
$$val_{D,s}(o.\mathtt{a} = e) = \{s''[\mathtt{a}(d_o) \leftarrow d_e] \mid (s', d_o) \in val_{D,s}(o) \wedge (s'', d_e) \in val_{D,s'}(e)\}$$
$$val_{D,s}(\mathtt{p}_1; \mathtt{p}_2) = \bigcup_{s' \in val_{D,s}(\mathtt{p}_1)} val_{D,s'}(\mathtt{p}_2)$$
$$val_{D,s}(\mathtt{if}(e)\ \{\mathtt{p}\}\ \mathtt{else}\ \{\mathtt{q}\}) = \begin{cases} val_{D,s',\beta}(\mathtt{p}), & (s', \mathit{True}) \in val_{D,s}(e) \\ val_{D,s',\beta}(\mathtt{q}), & (s', \mathit{False}) \in val_{D,s}(e) \\ \emptyset, & \text{otherwise} \end{cases}$$
$$val_{D,s}(\mathtt{while}(e)\{\mathtt{p}\}) = \begin{cases} \bigcup_{s_1 \in S_1} val_{D,s_1}(\mathtt{while}(e)\{\mathtt{p}\}) \text{ where } S_1 = val_{D,s'}(\mathtt{p}), \\ \qquad \text{if } (s', \mathit{True}) \in val_{D,s}(e) \\ \{s'\},\ \text{if } (s', \mathit{False}) \in val_{D,s}(e) \\ \emptyset,\ \text{otherwise} \end{cases}$$

Fig. 1. Definition of PL-DL semantic evaluation function (excerpt)

A *variable assignment* $\beta : \mathsf{LgV} \to \mathcal{D}_T$ maps a logical variable $x : T$ to its domain $\mathcal{D}_T$. 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 statement st to a set of states $val_{D,s}(st) \subseteq 2^S$; and (v) every expression $e : T$ to a set of pairs of state and value $val_{D,s,}(e) \subseteq 2^{S \times T}$. As PL is deterministic, all sets of states or state-value pairs have at most one element.

Fig. 1 shows an excerpt of the semantic definition of updates and programs, more definitions are in our technical report [9]. 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. 1. 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 $\{\mathtt{i} := \mathtt{j}\}\{\mathtt{j} := \mathtt{i}\}$ as an update. In the following context, if not stated otherwise, we use the upper-case letter $\mathcal{U}$ to denote this kind of "misused" update, compared to the real update that is denoted by a lower-case letter u. An update $\mathcal{U}$ could be the form of $\{u\}$ and $\{u_1\} \ldots \{u_n\}$.

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

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

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

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

Fig. 2. Selected sequent calculus rules (for more detail see [9,3])

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. 2) 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 2. We look at typical proof goals that arise during symbolic execution:

1. $\Gamma, \mathtt{i} > \mathtt{j} \Rightarrow \mathcal{U}[\mathtt{if\ (i>j)\ \{p\}\ else\ \{q\}}\ \omega]\phi$: Applying rule ifElse and simplification eliminates the `else` branch and symb. exec. continues with `p` ω.
2. $\Gamma \Rightarrow \{\mathtt{i} := \mathtt{c} \| \ldots\}[\mathtt{j = i;}\ \omega]\phi$ 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 [10].

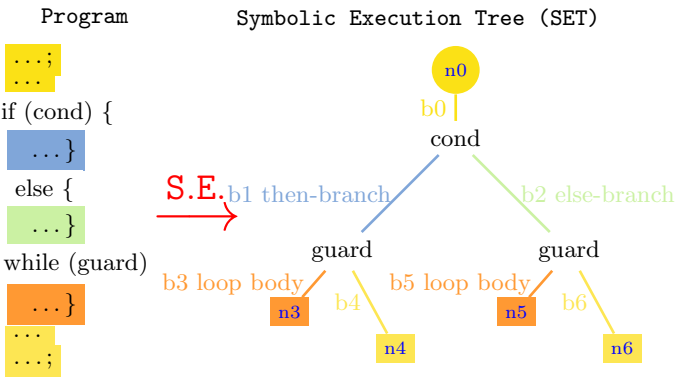

Fig. 3. Symbolic execution tree with loop invariant applied

3. $\Gamma \Rightarrow \{\texttt{o1.a} := \texttt{v1}\| \ldots\}[\texttt{o2.a = v2;}\ \omega]\phi$: After executing $\texttt{o2.a} = \texttt{v2}$, the *alias* is analyzed as follows: (i) if $\texttt{o2} = \texttt{null}$ is true the program does not terminate; (ii) else, if $\texttt{o2} = \texttt{o1}$ holds, the value of $\texttt{o1.a}$ in the update is overriden and the new update is $\{\texttt{o1.a} := \texttt{v2}\| \ldots \|\texttt{o2.a} := \texttt{v2}\}$; (iii) else the new update is $\{\texttt{o1.a} := \texttt{v1}\| \ldots \|\texttt{o2.a} := \texttt{v2}\}$. 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 $\texttt{p}$ following the sequent calculus rules is a *symbolic execution tree (SET)*, as illustrated in Fig. 3. Complete symbolic execution trees are finite acyclic trees whose root is labeled with $\Gamma \Longrightarrow [\texttt{p}]\phi, \Delta$ and no leaf has a $[\cdot]$ modality. W.l.o.g. we can assume that each inner node i is annotated by a sequent $\Gamma_i \Longrightarrow \mathcal{U}_i[\texttt{p}_\texttt{i}]\phi_i, \Delta_i$, where $\texttt{p}_\texttt{i}$ 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. We call a *sequential block* a maximal program fragment in an SET that is symbolically executed without branching. For instance, there are 7 sequential blocks in the SET on the right of Fig. 3.

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 (see [9]).

4.1 Weak Bisimulation Relation of Program

Definition 3 (Location sets, observation equivalence). *A* location set *Loc is a set containing program variables* x *and attribute expressions o.*a *with* a ∈ Attr *and o being a term of the appropriate sort.*

Given two states s_1, s_2 and a location set obs. A relation $\approx$: $Loc \times S \times S$ *is an* observation equivalence *iff for all D, β and ol ∈ 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 p (Fig. 1) is a state transformation. Executing p from a start state s results in a set of end states S', where S' is a singleton $\{s'\}$ if 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 p iff 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 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 p consists of simple statements. Execution of 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 $\mathtt{sSt}_i$ 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 4 (Observable and internal statement/transition). *Consider states s, s', a simple statement* sSt*, a transition relation* $\longrightarrow$*, where* $s \xrightarrow{\mathtt{sSt}} s'$*, and the observable locations obs; we call* sSt *an* observable statement *and* $\longrightarrow$ *an* observable transition*, iff for all D, β, there exists ol ∈ obs, and* $val_{D,s',\beta}(ol) \neq val_{D,s,\beta}(ol)$*. We write* $\xrightarrow{\mathtt{sSt}}_{obs}$*. Otherwise,* 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 sSt in the notion of the observable transition $\xrightarrow{\mathtt{sSt}}_{obs}$, 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 3. Given observable locations set *obs*={x, y}, the simple statement "`x = 1 + z;`" is observable, because x's value is reassigned (could be the same value). 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 may change the set of observable locations. Assume an observable transition $s \xrightarrow{\mathtt{sSt}}_{obs} s'$ changes the evaluation of some

location $ol \in obs$ in state s'. To continue with the execution of program $\mathtt{p}'$ from state s', the set of observable locations obs' in state s' should also contain the locations ol' that *read* the value of ol in some statement in $\mathtt{p}'$, because the change to ol can lead to a change of ol' at some later point in $\mathtt{p}'$.
Example 4. Consider obs={x, y} and program fragment "`z = x + y; x = 1 + z;`". `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' should also contain `z` after the execution of `z = x + y;` .

Definition 5 (Weak transition). *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 $\overset{\mathtt{sSt}}{\Longrightarrow}_{obs}$ is the composition of the relations $\Longrightarrow_{int}$, $\overset{\mathtt{sSt}}{\longrightarrow}_{obs}$ and $\Longrightarrow_{int}$: $s \overset{\mathtt{sSt}}{\Longrightarrow}_{obs} s'$ holds iff there are states s_1 and s_2 such that $s \Longrightarrow_{int} s_1 \overset{\mathtt{sSt}}{\longrightarrow}_{obs} s_2 \Longrightarrow_{int} s'$. The* weak transition $\overset{\widehat{\mathtt{sSt}}}{\Longrightarrow}_{obs}$ *represents either $\overset{\mathtt{sSt}}{\Longrightarrow}_{obs}$, 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 6 (Weak bisimulation for states). *Given two programs $\mathtt{p}_1, \mathtt{p}_2$ and observable locations obs, obs′, let $\mathtt{sSt}_1$ be a simple statement and s_1, s_1' two program states of $\mathtt{p}_1$, and $\mathtt{sSt}_2$ is a simple statement and s_2, s_2' are two program states of $\mathtt{p}_2$. A relation $\approx$ is a* weak bisimulation *for states iff $s_1 \approx_{obs} s_2$ implies:*

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

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

Definition 7 (Weak bisimulation for programs). *Let $\mathtt{p}_1, \mathtt{p}_2$ be two programs, obs, obs′ observable locations, and $\approx$ 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 $s_1^i \approx_{obs'} s_2^j$; (iii) for each state s_2^j there exists a state s_1^i such that $s_2^j \approx_{obs'} s_1^i$, where $0 \leq i \leq n$ and $0 \leq j \leq m$.

The above definition 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* [11]. The lemma Def. 7 to a *big-step semantics* [12].

Lemma 1. *Let $\mathtt{p}, \mathtt{q}$ be programs and obs the set of observable locations. If $\mathtt{p} \approx_{obs} \mathtt{q}$ then 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 8 (Weak bisimulation modality—syntax). *The* bisimulation modality $[\ \mathtt{p}\ \between\ \mathtt{q}\ ]@(obs, use)$ *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* $[\ \mathtt{p}\ \between\ \mathtt{q}\ ]@(obs, use)\phi$ *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 $l \in obs$ after executing a program $\mathtt{p}$. We approximate the set later by the set of all program variables in $\mathtt{p}$ that are used before being redefined.

Definition 9 (Used program variable). *A variable* $v \in \mathsf{PV}$ *is called* used by a program $\mathtt{p}$ *w.r.t. 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 heap locations and all used program variables of* $\mathtt{p}$ *w.r.t. 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*. E.g., variable $\mathtt{z}$ in Ex. 4 is a used variable. We formalize the semantics of the weak bisimulation modality:

Definition 10 (Weak bisimulation modality—semantics). *Given* $\mathtt{p}, \mathtt{q}$ *programs,* D, s, β*, and obs, use as above;* $val_{D,s,\beta}([\ \mathtt{p}\ \between\ \mathtt{q}\ ]@(obs, use)\phi) = tt$ *iff*

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

4.3 Sequent Calculus Rules for the Bisimulation Modality

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

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

Fig. 4 shows some extended sequent calculus rules, more are available in [9]. Unlike standard sequent calculus rules that are executed from root to leaves, sequent rule application for the bisimulation modality consists of two phases: In the first phase, the source program p is evaluated 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*

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

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

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

(with b boolean variable.)

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

Fig. 4. A collection of sequent calculus rules for program transformation

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. 3. In the second phase, we synthesize the target program `q` and used variable set `use` from $\texttt{q}_i$ and $\texttt{use}_\texttt{i}$ by applying the rules in a leaves-to-root manner. One starts with a leaf node and apply the emptyBox rule, then stepwise generates the program within its sequential block, e.g., b3,. . . , b6 in Fig. 3. These are combined by rules corresponding to statements that contain a sequential block, such as loopInvariant (containing b3 and b4). One continues with the sequential block containing the compound statements, e.g., b2, until the root is reached. Note that the order of processing the sequential blocks matters, for instance, the program for the sequential block b4 must be generated before that for b3, because the observable locations in node n3 depend on the used variable set of b4 according to the loopInvariant rule.

Lemma 2. *The extended sequent calculus rules are sound. (For the proof see [9])*

5 Optimization

Sect. 4.2 introduced an approach to program simplification based on the extended sequent calculus rules. The generated program consists only of simple statements and is optimized to a certain degree, because the used variable set avoids generating unnecessary statements. 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, it can be used for further optimization.

5.1 Update Simplification

Within a sequential block, after application of sequent rules (e.g., assignment), we often obtain an update $\mathcal{U}$ of the form $\{u_1\} \dots \{u_n\}$. It can be simplified into a single update $\{u\}$, namely the *normal form* (NF) of update.

Definition 11 (Normal form of update). *An update is in* normal form, *denoted by* $\mathcal{U}^{nf}$, *if it has the shape* $\{u_1 \| \dots \| u_n\}$, $n \geq 0$, *where each* u_i *is an elementary update and there is no conflict between* u_i *and* u_j *for any* $i \neq j$.

The normal form of an update $\mathcal{U} = \{u_1\} \dots \{u_n\}$ can be achieved by applying a sequence of *update simplification* steps. Soundness of these rules and that they achieve normal form are proven in [13]. The update rules are reproduced in [9].

Like elementary updates, updates in normal form are in SSA. It is easy to maintain normal form of updates in a sequential block when applying the extended sequent calculus rules of Fig. 4. This can be used for further optimization of the synthesized 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 5. Consider the program "`i = j + 1; j = i; i = j + 1;`". After executing the first two statements and 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 `j` + 2 and `j` has value `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. 4 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:

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

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

However, this rule is not sound. If we continue Ex. 5 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 `j` + 3 instead of `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.

Definition 12 (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 13 (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 SNF update that has only a core part.

Example 6 (SNF update).

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

To compute the SNF of an update, we need two more rules:

- (*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 $\mathtt{x}^0 \notin pv$

Lemma 3. *The associativity rule and introducing auxiliary rule are sound.*

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 introduce auxiliary rule to introduce $\{l^0 := l\}$, then compute the new auxiliary update and core update.

5.2 Extended Sequent Calculus Rules Involving Updates

With the help of the SNF of an update, the assignment rule becomes:

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

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 auxAssignment rule

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

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

is used. I.e., the auxiliary assignments are always generated at the start of a sequential block. Most other rules are obtained by replacing $\mathcal{U}$ with $\mathcal{U}^{snf}$, see [9].

Example 7. We demonstrate that the program from Ex. 5 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.

Remark. The program is first synthesized within a sequential block and then constructed. The SNF updates used in the above rules belong to the current sequential block. An execution path may contain several sequential blocks. We keep the SNF update for each sequential block without simplifying them further into a bigger SNF update for the entire execution path. E.g. in Fig. 3, the execution path from node n0 to n4 involves 3 sequential blocks b0, b1 and b4. When we synthesize the program in b4, 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 the b4 when generating the program for b4, so in the above rules, $\mathcal{U}^{snf}$ refers to $\mathcal{U}_4^{snf}$ and the other SNF updates are omitted.

Lemma 4. *The extended sequent calculus rules involving updates are sound.*

6 Related Work

JSpec [14] 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 [15], 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 [16], but they need to encode the correctness properties into a logic programming language.

Verifying Compiler [17] 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 [18], 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 [19] 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 [20], 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.

7 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: Leavens, G.T., O'Hearn, P., Rajamani, S.K. (eds.) VSTTE 2010. LNCS, vol. 6217, pp. 40–54. Springer, Heidelberg (2010)
2. Baumann, C., Beckert, B., Blasum, H., Bormer, T.: Lessons learned from microkernel verification – specification is the new bottleneck. In: SSV. EPTCS, vol. 102, pp. 18–32 (2012)
3. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software. LNCS (LNAI), vol. 4334. Springer, Heidelberg (2007)
4. Bubel, R., Hähnle, R., Ji, R.: Program specialization via a software verification tool. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2010. LNCS, vol. 6957, pp. 80–101. Springer, Heidelberg (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), 385–394 (1976)

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), 32–54 (2005)
8. Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. MIT Press (2000)
9. Ji, R., Hähnle, R., Bubel, R.: Program transformation based on symbolic execution and deduction, technical report (2013)
10. Bubel, R., Hähnle, R., Ji, R.: Interleaving symbolic execution and partial evaluation. In: de Boer, F.S., Bonsangue, M.M., Hallerstede, S., Leuschel, M. (eds.) FMCO 2009. LNCS, vol. 6286, pp. 125–146. Springer, Heidelberg (2010)
11. Plotkin, G.D.: A structural approach to operational semantics. J. Log. Algebr. Program. 60-61, 17–139 (2004)
12. Kahn, G.: Natural semantics. In: Brandenburg, F.J., Wirsing, M., Vidal-Naquet, G. (eds.) STACS 1987. LNCS, vol. 247, pp. 22–39. Springer, Heidelberg (1987)
13. Rümmer, P.: Sequential, parallel, and quantified updates of first-order structures. In: Hermann, M., Voronkov, A. (eds.) LPAR 2006. LNCS (LNAI), vol. 4246, pp. 422–436. Springer, Heidelberg (2006)
14. Schultz, U.P., Lawall, J.L., Consel, C.: Automatic program specialization for Java. ACM-TPLS 25(4), 452–499 (2003)
15. Ruf, E.S.: Topics in online partial evaluation. PhD thesis, Stanford University, Stanford, CA, USA, UMI Order No. GAX93-26550 (1993)
16. Hatcliff, J., Danvy, O.: A computational formalization for partial evaluation. Mathematical Structures in Computer Science 7(5), 507–541 (1997)
17. Hoare, T.: The verifying compiler: A grand challenge for computing research. J. ACM 50, 63–69 (2003)
18. Leroy, X.: Formal verification of a realistic compiler. CACM 52(7), 107–115 (2009)
19. Barthe, G., Crespo, J.M., Kunz, C.: Relational verification using product programs. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 200–214. Springer, Heidelberg (2011)
20. Barthe, G., Crespo, J.M., Gulwani, S., Kunz, C., Marron, M.: From relational verification to SIMD loop synthesis. In: PPOPP, pp. 123–134. ACM (2013)