Modular, Correct Compilation with Automatic Soundness Proofs*

Dominic Steinhöfel[0000−0003−4439−7129] and Reiner Hähnle[0000−0001−8000−7613]

TU Darmstadt, Dept. of Computer Science, Darmstadt, Germany
{steinhoefel,haehnle}@cs.tu-darmstadt.de

Abstract. Formal verification of compiler correctness requires substantial effort. A particular challenge is lack of modularity and automation. Any change or update to the compiler can render existing proofs obsolete and cause considerable manual proof effort. We propose a framework for automatically proving the correctness of compilation rules based on simultaneous symbolic execution for the source and target language. The correctness of the whole system follows from the correctness of each compilation rule. To support a new source or target language it is sufficient to formalize that language in terms of symbolic execution, while the corresponding formalization of its counterpart can be re-used. The correctness of translation rules can be checked automatically. Our approach is based on a reduction of correctness assertions to formulas in a program logic capable of symbolic execution of *abstract* programs. We instantiate the framework for compilation from Java to LLVM IR and provide a symbolic execution system for a subset of LLVM IR.

1 Introduction

Writing correct programs is hard. All the more painful it is, when a program with semantically correct source code does not execute as expected due to a compiler bug. Happily, this is not a common experience to programmers. Still, commonly used compilers, like any complex software product, *do* contain bugs [12]; and for safety-critical settings, one might just not want to take the chance. Testing compilers can only show the presence, but not the absence of bugs. A more ambitious undertaking is the construction of a *verified compiler*, as carried out in the CompCert project [13], Jinja [10] or the more recent CakeML [18]. In these approaches, the compiler is programmed in the executable language fragment of an interactive proof assistant and proven correct relative to a mechanized semantics of the source and target language. Such interactive approaches involve a significant amount of work for the construction of proof scripts alone (44% of all code in CompCert [13], for example). In addition, it is not easy to keep proofs modular in the sense that arguing the correctness of compilation of one

* This work was funded by the Hessian LOEWE initiative within the Software-Factory 4.0 project. **The final authenticated version is available online at** https://doi.org/10.1007/978-3-030-03418-4_25.

$$\textsf{assgnPVTransl}\ \frac{(\mathcal{U} \circ (\mathrm{x} := \mathrm{y}), C, \pi\ \omega \Vdash q^n)@(obs)}{(\mathcal{U}, C, \pi\ \texttt{x=y;}\ \omega \Vdash \left(q \lhd_n \left(\begin{array}{l}\texttt{\%0 = load i32, i32* \%y}\\ \texttt{store i32 \%0, i32* \%x}\end{array}\right)\right)^{(n)})@(obs)}$$

Fig. 1: Translation rule for a variable assignment

language construct is not dependent on another. Here we lay down a theoretical framework of a *modular*, verified compilation approach that avoids these issues: (1) In compiler verification, local changes to syntax or semantics of source or target language as well as to compilation rules, are likely to affect the correctness proof globally. We avoid this by a rule-driven approach wherein local changes are confined to a single rule. (2) Proofs in assistants like Coq and Isabelle/HOL are mostly interactive. When adding support for new source elements or another front- or backend, new interactive proofs have to be constructed for the translation and existing ones might have to be adapted. In our rule-driven approach it is sufficient to *add* new rules whose correctness proofs, moreover, are automatic.

The paper is structured as follows. Next we give an informal overview of our approach. In Sect. 3, we introduce our logical foundation and Symbolic Execution semantics. Sect. 4 presents our formalization of a subset of LLVM IR. Central definitions and correctness properties of translation rules, as well as example rules for translation of Java to LLVM IR, are given in Sect. 5. In Sect. 6, we describe how to handle loops. Finally, we discuss related work in Sect. 7 and present a conclusion and outlook in Sect. 8.

2 Overview

Our approach is based on Symbolic Execution (SE) [5]. The compiler is defined by a set of translation rules, each of which realizes *simultaneous* SE of an aspect of the source and target language. An example of a rule assgnPVTransl for variable assignment is shown in Fig. 1. The rule expresses that the highlighted code fragments in Java and LLVM IR have the same *effect* (on a symbolic set of observable variables *obs*), namely setting the value of variable x to the value of y in an existing (symbolic) store $\mathcal{U}$. An alternative interpretation is that the Java fragment can be compiled in a behavior-preserving manner to the LLVM IR fragment. Compilation works in two phases: We symbolically execute the source program, then apply translation rules to the resulting Symbolic Execution Tree in a leaves-to-root manner to obtain the compiled program. Besides the high modularity of our system due to its rule-based nature, an important advantage is that we can prove the correctness of rules like assgnPVTransl *automatically*. Prerequisite is a formalization of the source *and* the target language in terms of SE rules.[1] The program fragments occurring in translation rules are typically *abstract*, because they contain placeholders for subprograms (e.g., the guard and

[1] Compiler verification with interactive proof assistants requires this as well, even separate formalizations for the target and source languages.

body of an `if` statement), as well as premises with assumptions about them. We obtain correctness results for translation rules by reflecting correctness assertions to a program logic capable of symbolically executing the *abstract* source and target programs of a rule. The resulting assertions can be proven by an automatic program verifier.

The logical basis of our approach is a new formalization of SE for abstract programs based on a notion of concretization and a definition of soundness of SE transition relations. We instantiate our framework for compilation from Java to LLVM IR. This choice is motivated by the following considerations: For Java, we can build on a mature SE system based on Java Dynamic Logic (JavaDL) [1] which suits our needs very well. LLVM IR [11] is the intermediate language of the state-of-the-art LLVM optimizing compilation framework, which is employed in a variety of commercial and open source products as well as academic research prototypes. The language comes with attractive properties: it is typed and uses an unbounded number of temporary registers instead of a stack. Finally, as far as we know, there is no compiler from Java to LLVM IR currently maintained.

Extending our compiler framework, for example, with a new target language, is achieved by providing that language's formalization in terms of SE rules (which, as a byproduct, can also be used for different purposes, such as program verification). The compilation rules defined below are proven without assistance. The compiler is, therefore, *correct-by-construction*. Our main contribution is a theoretical framework consisting of (1) a semantic foundation for SE of abstract programs, (2) a new, partial formalization of LLVM IR, and (3) theoretical results about proving the correctness of translation rules, illustrated by the compilation of Java to LLVM IR.

3 Program Logic and Symbolic Execution

Symbolic Execution [5] is a popular static program analysis technique that treats inputs to a program as abstract symbols. Whenever the execution depends on the concrete value of a symbolic variable, it performs a case distinction, following each possible branch individually. The outcome is a Symbolic Execution Tree (SET) whose root is labeled with the program under execution. Inner nodes are constructed by individual SE steps. The version of SE used in this paper consists of a static symbolic interpreter embedded in a logical framework—Dynamic Logic (DL): an extension of typed first-order logic for expressing assertions about program behavior, inspired by features of JavaDL [1, Chapter 3], a DL for Java.

The principles of our logic are general enough to be re-used for different languages, such as C# or Java bytecode. The translation rule in Fig. 1 highlights a central concept of our logic: The assignment of the variable $\mathtt{y}$ to $\mathtt{x}$ corresponds to a state *update* $\mathtt{x} := \mathtt{y}$. Updates describe state changes resulting from symbolic program execution. *Elementary* updates $\mathtt{x} := t$ syntactically represent the state changes where the variable $\mathtt{x}$ attains the value of the term t. Updates can be combined to *parallel* updates $\mathtt{x} := t_1 \,||\, \mathtt{y} := t_2$, and be *applied* to terms, formulas and other updates. We write $\{\mathcal{U}\}t$ for the application of $\mathcal{U}$ to term t.

Semantically, t is evaluated in the state represented by $\mathcal{U}$. The "empty update" is denoted by *Skip*. Our term language contains conditional terms; for instance, $(\mathit{if}\,(\mathtt{i} > \mathtt{j})\,\mathit{then}\,(\mathtt{i})\,\mathit{else}\,(\mathtt{j})) \geq 0$ means that the maximum of `i` and `j` is positive.

The semantics of our logic is based on a *valuation function* $\mathit{val}\,(K, \sigma|\cdot)$, where K is a structure and σ a concrete program state. The set of all concrete states σ assigning domain values to (program) variables is denoted by $\mathcal{S}_{concr}$. The valuation function assigns to *formulas* a truth value "true" or "false". We write $K, \sigma \models \varphi$ for $\mathit{val}\,(K, \sigma|\varphi) = \text{true}$ (similarly for $K, \sigma \not\models \varphi$). To *terms*, it assigns a value of the domain of the term, to *updates* a state transformer $\mathcal{S}_{concr} \to \mathcal{S}_{concr}$ and to *programs* a function $\mathcal{S}_{concr} \to 2^{\mathcal{S}_{concr}}$, where the output, for deterministic programs, is the empty set if the program does not terminate, or a singleton otherwise. A term domain we often use is that of booleans, which consists of the values TRUE and FALSE; we employ a function $\mathit{for2bool}(\varphi) \coloneqq \mathit{if}\,(\varphi)\,\mathit{then}\,(\text{TRUE})\,\mathit{else}\,(\text{FALSE})$ to convert formulas to booleans. As an example for the valuation of an update, consider "$\mathtt{x} := \mathtt{y}$": The result of the valuation $\mathit{val}\,(K, \sigma|\mathtt{x} := \mathtt{y})\,(\sigma')(\mathtt{z})$ is either $\sigma(\mathtt{y})$, if $\mathtt{z} = \mathtt{x}$, or $\sigma'(\mathtt{z})$ otherwise.

For formulating assertions about program behavior, DL contains *modalities*: A formula $[p]\varphi$ expresses the partial correctness property that *if* p terminates, then the formula φ holds. Semantically, $\mathit{val}\,(K, \sigma|[p]\varphi)$ evaluates to true if for each $\sigma' \in \mathit{val}\,(K, \sigma|p)\,(\sigma)$, it holds that $\mathit{val}\,(K, \sigma'|\varphi) = \text{true}$. For full details on JavaDL for Java and complete definitions, we refer to [1]. Here we focus on the aspects of the logic that are specific to LLVM IR and SE: An LLVM IR program $q^{(n)}$ is an instruction list with an *instruction pointer* $n \in \mathbb{Z}$. If the pointer is omitted, we assume the default "0" pointing to the first instruction in the list. If n is negative or exceeds the number of instructions in q, the program has already been exited and its valuation is the identity function; this corresponds to an "empty" program for Java. We also write "`_`" in this case (for both languages).

The nodes of an SET are *symbolic execution states* $(\mathcal{U}, C, p)$ consisting of a *symbolic store* $\mathcal{U}$, a *path condition* C and a *program counter* p. In our framework, $\mathcal{U}$ is an update, C a set of quantifier-free formulas; p can either be a Java program $\pi\ \mathit{stmt}\ \omega$ or an LLVM IR program $q^{(n)}$ with instruction pointer as defined above. In case of Java, the *active statement* stmt of a program is the next to be executed. The *prefix* π contains opening braces, labels etc., and ω the remaining program.

We use the associative operator $\mathcal{U}_1 \circ \mathcal{U}_2 \coloneqq \mathcal{U}_1 \,||\, \{\mathcal{U}_1\}\mathcal{U}_2$ for combining sequential updates. SE aims to transform a program p into a set of states with empty program counters. The symbolic stores and path conditions of these states then together describe the effect of p. Consider the (Java) SE state $(\mathtt{y} := \mathtt{z}, \emptyset, \mathtt{x=y;})$, which is in a single SE step transformed into $(\mathtt{y} := \mathtt{z} \circ \mathtt{x} := \mathtt{y}, \emptyset, \_)$. By definition of $\circ$, this equals $(\mathtt{y} := \mathtt{z} \,||\, \{\mathtt{y} := \mathtt{z}\}(\mathtt{x} := \mathtt{y}), \emptyset, \_)$, which can be simplified to $(\mathtt{y} := \mathtt{z} \,||\, \mathtt{x} := \mathtt{z}, \emptyset, \_)$. With $\mathbb{S}_{SE}$ we denote the set of all SE states, Upd is the set of updates and Fml^{qf} the set of quantifier-free formulas. Semantically, an SE state represents (possibly infinitely many) concrete states. Given a concrete initial state, we can *concretize* a *symbolic* execution state to a *concrete* state.

The union of all those *concretizations* for all concrete initial states provides us with a complete description of the semantics of an SE state.

Definition 1 (Concretization Function). *The* concretization function $concr : \mathbb{S}_{SE} \times \mathcal{S}_{concr} \to 2^{\mathcal{S}_{concr}}$ *maps an SE state* $(\mathcal{U}, C, p) \in \mathbb{S}_{SE}$ *and a concrete state* $\sigma \in \mathcal{S}_{concr}$ *(1) to* $\emptyset$ *if* $K, \sigma \not\models C$*, and else (2) to the set* $val\,(K, val\,(K, \sigma|\mathcal{U})\,(\sigma)|p)\,(\sigma)$ *(which, as Java is deterministic, is either a singleton if* p *terminates, or empty).*

Example 1. The Java program in the SE state $s = (Skip, \emptyset,$ `if (x<0) x=-x;`$)$ computes the absolute value of variable `x`. The leaves of the corresponding SET are $(Skip, \{\mathrm{x} \geq 0\}, \_)$ and $(\mathrm{x} := -\mathrm{x}, \{\mathrm{x} < 0\}, \_)$. Intuitively, s represents all concrete states where `x` has a positive value. Given $\sigma \in \mathcal{S}_{concr}$ with $\sigma(\mathrm{x}) = -17$, $concr(s, \sigma)$ is the singleton set of a state mapping `x` to 17 and all other variables to their values in σ. For each $\sigma' \in \bigcup_{\sigma \in \mathcal{S}_{concr}} concr(s, \sigma)$ it holds that $\sigma'(\mathrm{x}) \geq 0$.

An *SE transition relation* $\delta \subseteq \mathbb{S}_{SE} \times \mathbb{S}_{SE}$, which defines how to obtain SETs of SE states, in our framework is composed of a set of *restricted* SE rules. The restriction is that each rule may only add to (and not remove or alter) the symbolic stores (by update concatenation) and path conditions (by set union). The resulting programs are not syntactically restricted.

A sound SE step has to transform programs into corresponding changes in the symbolic store, as in the Java part of the rule assgnPVTransl in Fig. 1: The assignment `x=y` is transformed into an update $\mathrm{x} := \mathrm{y}$ that leaves the concretizations of the state unchanged. Case distinctions have to be *exhaustive* and *disjoint*. A rule for an `if` statement creates two states. In the first, the guard is added to the path condition, and the program counter contains the *then* part; in the second, the path condition contains the *negation* of the guard, and the program counter the *else* part. A concrete state satisfying the path condition of the initial SE state will *either* satisfy the path condition of the first *or* the second successor, since it satisfies the guard or its negation, but not both. The following formal definition for the soundness of SE transition relations uses three projection functions, $\cdot_{store}$, $\cdot_{path}$ and $\cdot_{pc}$, to obtain the symbolic store, path condition and program counter of an SE state. Subsequently, we assume SE transition relations to be sound.

Definition 2 (Soundness of Symbolic Execution). *An SE transition relation* $\delta \subseteq \mathbb{S}_{SE} \times \mathbb{S}_{SE}$ *is called* sound *iff (1) for each input-output pair* $(i, o) \in \delta$ *and* $\sigma \in \mathcal{S}_{concr}$ *with* $K, \sigma \models o_{path}$*, it holds that* $concr(i, \sigma) = concr(o, \sigma)$*, and (2) for each* $i \in \mathbb{S}_{SE}$ *for which there is an outgoing transition in* δ*, and* $\sigma \in \mathcal{S}_{concr}$ *with* $K, \sigma \models i_{path}$*, there is* exactly one $o \in \mathbb{S}_{SE}$ *such that* $(i, o) \in \delta$ *and* $K, \sigma \models o_{path}$*.*

To handle placeholders in translation rules, we introduce *abstract programs*, which contain symbols $\widehat{P}$ from a set $AbsP$ of abstract program symbols. Each $\widehat{P}$ represents an *equivalence class* of programs with the same effect. To formally define SE of abstract programs we use *abstract block contracts* that generalize the concept of *abstract operation contracts* [4, 7]. A *block contract* for a code

```
1 %1 = load i32, i32* %x ; <label>:0
2 %2 = icmp slt i32 %1, 0
3 br i1 %2, label %3, label %6
4 %4 = load i32, i32* %x ; <label>:3
5 %5 = sub nsw i32 0, %4
6 store i32 %5, i32* %x
7 br label %6
8 ; ...                  ; <label>:6
```

Listing 1: Absolute of variable x

block bl is a pair $(\mathcal{U}^a_{bl}, C_{bl})$, where $\mathcal{U}^a_{bl}$ is an update representing the assignable clause of bl ($\mathcal{U}^a_{bl}$ assigns fresh constants to variables assigned in the block, and similarly anonymizes the heap), and C_{bl} is a set of formulas representing bl's postcondition. An SE application of the block contract rule transforms an SE state $(\mathcal{U}, C, bl;\ p)$ to $(\mathcal{U} \circ \mathcal{U}^a_{bl}, C \cup \{\{\mathcal{U}^a_{bl}\}C_{bl}\}, p)$, which is sound provided that bl respects its contract. While in the block contract rule bl is a concrete program and $(\mathcal{U}^a_{bl}, C_{bl})$ a concrete contract, *abstract* contracts admit *placeholders* in post conditions and assignable clauses. We lift SE to *abstract execution*, because now, in addition, bl can be an abstract program symbol $\widehat{P}$:

Definition 3 (Abstract Execution). *Let $C_{\widehat{P}}$, $\mathcal{U}^a_{\widehat{P}}$ be fresh Skolem constants representing unknown postconditions and assignable clauses.* Abstract execution *transforms an SE state of the form $(\mathcal{U}, C, \widehat{P}\ p)$ to $(\mathcal{U} \circ \mathcal{U}^a_{\widehat{P}}, C \cup \{\{\mathcal{U}^a_{\widehat{P}}\}C_{\widehat{P}}\}, p)$.*

Observation 1 *Abstract execution of $\widehat{P}$ with the block contract rule is sound: any substitution of a concrete code block bl^c for bl, a concrete post condition C^c_{bl} for C_{bl}, and a concrete update $\mathcal{U}^c_{bl}$ for $\mathcal{U}^a_{bl}$, where bl^c respects C^c_{bl} and $\mathcal{U}^c_{bl}$, yields a sound concrete SE transition.*

Example 2. Consider the Java SE state $(\texttt{i} := 0, \emptyset, \widehat{P}\ \texttt{i=17;})$ containing the abstract program $\widehat{P}$ as active statement. Abstract execution of $\widehat{P}$ results in $((\texttt{i} := 0) \circ \mathcal{U}^a_{\widehat{P}}, C_{\widehat{P}}, \texttt{i=17;})$, which evaluates to $((\texttt{i} := 0) \circ \mathcal{U}^a_{\widehat{P}} \circ (\texttt{i} := 17), C_{\widehat{P}}, \_)$. We can further simplify this state to $(\mathcal{U}^a_{\widehat{P}} \circ (\texttt{i} := 17), C_{\widehat{P}}, \_)$ since i is overwritten in the last update. Based on the execution result, we can prove the assertion that i is 17 after each execution of the abstract program (we ignore the possibility of thrown exceptions for this example). This corresponds to showing the validity of the DL formula $\{\texttt{i} := 0\}[\widehat{P}\ \texttt{i=17;}]\texttt{i} \doteq 17$, which simplifies to $C_{\widehat{P}} \rightarrow \{\mathcal{U}^a_{\widehat{P}} \circ (\texttt{i} := 17)\}\texttt{i} \doteq 17$ after SE, and further to $C_{\widehat{P}} \rightarrow \{\texttt{i} := 17\}\texttt{i} \doteq 17$, which is true. A possible concrete substitution for the abstract elements is $\widehat{P} \coloneqq (\texttt{x=12;y=0;})$, $C_{\widehat{P}} \coloneqq \{\texttt{x} \doteq 12\}$ and $\mathcal{U}^a_{\widehat{P}} \coloneqq (\texttt{x} := c_{\texttt{x}} \,||\, \texttt{y} := c_{\texttt{y}})$ for fresh $c_{\texttt{x}}, c_{\texttt{y}}$.

4 Formalizing LLVM IR

LLVM IR [11] is a typed, low-level, SSA-based, RISC-like virtual instruction set. Unlike most RISC instruction sets, LLVM does not have a fixed set of named

registers and a stack, but an *infinite set of temporaries* %0, %1, etc. which have to be assigned sequentially. Named registers %x, %y etc. can also be introduced. In our model, named registers represent program variables in the Java sense. LLVM IR code is structured into *basic blocks*, sequences of instructions ending in one terminator instruction (like a branch or return). If a basic block does not have an explicit label, it is assigned the next free temporary register as label. Labelling instructions *within* a basic block is not possible, only the start of a basic blocks can be a jump target. Listing 1 shows an LLVM IR program computing the absolute of a 32-bit integer variable (**i32**) x, whose address is contained in register %x. The first **load** instruction loads the value at address %x into register %1. The **icmp slt** ("signed less than") instruction compares the result with 0, setting %2 to 1 (i.e., TRUE) iff x is negative. Line 3 performs a case distinction via a labeled **br** instruction: If the bit (**i1**) in register %2 is 1 (the result of the comparison is that x is negative), the program continues with the code inverting x at label %3. Otherwise, we directly skip to the end of the program at label %6. In line 4, the program again loads the value of x, which is at line 5 subtracted (**sub**) from 0 (the **nsw** stands for "no signed wrap"). The **store** instruction at line 6 stores the result of that substraction (the inverted, now positive variable x) at address %x; the unconditional **br** instruction at line 7 causes the start of a new basic block at line 8. Comments start with a semicolon. The comments in the listing indicate the implicit label registers: their type is a jump address.

We define a *statement injection function* $\lhd_n$: The result of $q \lhd_n q'$ is a program where at the n-th position of q, the program q' has been inserted, such that the first instruction of q' is the n-th in the result. The function updates temporary registers to maintain their sequential order. Figure 2 shows some SE rules for LLVM IR. They are read bottom-up like sequent calculus rules. The rules llvmStoreInt and llvmLoadBool illustrate the treatment of named vs. anonymous registers. For instance, a **store** instruction writing the value at register %i into *address* %x is executed by setting the *variable* x to the value %i in the symbolic store and incrementing the instruction pointer (the **load** of a boolean works similarly). Rule llvmLtComp executes a less-than comparison of the contents of two registers %k and %l. The boolean expression *for2bool*(%k < %l) evaluates to TRUE iff the formula %k < %l evaluates to true. In those three rules, SE amounts to extending the update and increasing the instruction pointer by one. Rule llvmCondBreakUnrestr is an example for a *branching* rule (we omit the—straightforward—unconditional variant). Since the value of %i is in general symbolic (and might be TRUE *or* FALSE), we perform a case distinction, following the possible branches independently. In the first case, we assume that %i is TRUE and continue at the position n_1 of the then-label %j, and analogously in the second case. For programs with backjumps, i.e. $n_i < n$, SE might not terminate, because of this rule. In Sect. 6 we introduce a method for handling loops which ensures termination.

$$\textsf{IlvmStoreInt}\ \frac{(\mathcal{U} \circ (\mathrm{x} := \%i), C, (q \lhd_n (\texttt{store i32 \%}i\texttt{, i32* \%x}))^{(n+1)})}{(\mathcal{U}, C, (q \lhd_n (\texttt{store i32 \%}i\texttt{, i32* \%x}))^{(n)})}$$

$$\textsf{IlvmLoadBool}\ \frac{(\mathcal{U} \circ (\%i := \mathrm{x}), C, (q \lhd_n (\%i\texttt{ = load i1, i1* \%x}))^{(n+1)})}{(\mathcal{U}, C, (q \lhd_n (\%i\texttt{ = load i1, i1* \%x}))^{(n)})}$$

$$\textsf{IlvmLtComp}\ \frac{(\mathcal{U} \circ (\%i := \mathit{for2bool}(\%k < \%l)), C, (q \lhd_n (\%i\texttt{ = icmp sle i32 \%}k\texttt{, \%}l))^{(n+1)})}{(\mathcal{U}, C, (q \lhd_n (\%i\texttt{ = icmp sle i32 \%}k\texttt{, \%}l))^{(n)})}$$

$$\textsf{IlvmCondBrUnrestr}\ \frac{\begin{array}{c}(\mathcal{U}, C \cup \{\%i \doteq \mathrm{TRUE}\}, (q \lhd_n (\texttt{br i1 \%}i\texttt{, label \%}j\texttt{, label \%}k))^{(n_1)}) \\ (\mathcal{U}, C \cup \{\%i \doteq \mathrm{FALSE}\}, (q \lhd_n (\texttt{br i1 \%}i\texttt{, label \%}j\texttt{, label \%}k))^{(n_2)})\end{array}}{(\mathcal{U}, C, (q \lhd_n (\texttt{br i1 \%}i\texttt{, label \%}j\texttt{, label \%}k))^{(n)})}$$

where n_1 and n_2 are the positions of the labels $\%j$ and $\%k$, respectively.

Fig. 2: Some LLVM IR SE rules

5 Program Translation

We follow the approach of *rule-based compilation* [2,3] and additionally base our system on *heavyweight symbolic execution* [17]. The advantages of rule-based over monolithic systems consist of a higher degree of abstraction and better modularity [3]. The latter not only increases reusability, but is also instrumental in defining the modular correctness notion of our framework. The basics of our compilation process were already sketched in earlier work [9]: First we symbolically execute the source program, and then apply to the resulting SET, starting from its leaves, a set of translation rules. The result is an SET consisting of *dual* SE states, each of which contains equivalent programs in the source and target language. The root of the tree contains the compiled program. Our rules are not mere transformation rules, but rules for *simultaneous symbolic execution*, where each element of the source language is associated with an SE rule. In the following we establish basic notions for simultaneous SE and state the main result for proving the correctness of translation rules. Then we exemplarily define some rules for the translation from Java to LLVM IR and illustrate our approach along the translation of an `if` statement.

Definition 4 (Dual SE States and Transition Relations). *Given an update $\mathcal{U}$, a path condition C, a Java program p and an LLVM IR program $q^{(n)}$, and a set of observable variables obs, we call the triple $(\mathcal{U}, C, p \dashv\vdash q^{(n)})@(obs)$ a* dual symbolic execution state. *The set of all dual SE states is denoted by $\mathbb{S}^{\mathsf{d}}_{SE}$. A* Dual SE Transition Relation (DSETR) *is a relation $\delta^{\mathsf{d}} \subseteq \mathbb{S}^{\mathsf{d}}_{SE} \times \mathbb{S}^{\mathsf{d}}_{SE}$.*

We call a dual SE state *valid* if the source and target program have the same *observable semantics* in the states defined by the symbolic store and path condition, i.e. their concretizations coincide on the observable variables. Formally:

$$\textsf{ruleName}\quad \frac{(\mathcal{U}\circ\mathcal{U}_1, C\cup C_1, p_1 \dashv\vdash q^{(n_1)})@(obs_1) \quad \cdots \quad (\mathcal{U}\circ\mathcal{U}_m, C\cup C_m, p_m \dashv\vdash q^{(n_m)})@(obs_m)}{(\mathcal{U}, C, p \dashv\vdash q^{(n)})@(obs_1\cup\cdots\cup obs_m)}$$

Fig. 3: Schematic translation rule

Definition 5 (Validity of Dual SE States). *A dual SE state* $(\mathcal{U}, C, p \dashv\vdash q^{(n)})@(obs)$ *is* valid *iff the concretizations of* $s_p = (\mathcal{U}, C, p)$ *and* $s_q = (\mathcal{U}, C, q^{(n)})$ *coincide on all* $pv \in obs$*, i.e.* p *and* $q^{(n)}$ *either both do not terminate, or they terminate and, where* $\sigma \in \mathcal{S}_{concr}$ *such that* $K, \sigma \models C$*, and* σ_p *and* σ_q *are the concrete states in the singleton sets* $concr(s_p, \sigma)$ *and* $concr(s_q, \sigma)$*,* $\sigma_p(pv) = \sigma_q(pv)$*.*

Definition 6 (Soundness of DSETRs). *A DSETR* δ^{d} *is* sound *iff the validity of each dual SE state* i *is implied by the validity of* all *of its output states, i.e. of all states in* $\{o : (i, o) \in \delta^{\mathsf{d}}\}$*.*

This definition of soundness of DSETRs is suitable to ensure that SE rules preserve validity (read "top-down"). Together, Defs. 5, 6 imply: if, starting from a dual SE state $s = (\mathcal{U}, C, p \dashv\vdash q^{(n)})@(obs)$ and by applying a sound DSETR, we can derive a set of final states that all have the form $(\mathcal{U}', C', \_ \dashv\vdash \_)@(obs')$ for some $\mathcal{U}'$, C' and obs', then s is valid, which means that the programs p and $q^{(n)}$ have the same observable effects when started in a state satisfying $\mathcal{U}$ and C.

A DSETR can be defined in terms of rules for elements of the input programming languages. Hence, soundness of the resulting DSETR can be established by checking whether all *rules* meet the requirement of Def. 6. Figure 3 shows a schematic representation of such a rule. It can be viewed in two ways. First, as a rule for simultaneous symbolic execution, which is read "bottom-up": We transform the dual SE state in the conclusion to simpler states until the programs have been fully executed. Second, as a translation rule: Program p can be translated to program $q^{(n)}$ (relative to $\mathcal{U}, C$), where both programs may be abstract. The requirements on the placeholders and the corresponding symbolic stores $\mathcal{U}_i$ and path conditions C_i are defined in the premisses. Figure 3 also illustrates the restrictions we impose on dual SE rules, which correspond to those of their non-dual counterparts: Symbolic stores and path conditions may only be *extended*, but not altered otherwise ($\mathcal{U}$ and C are still present in all premises).

According to Def. 6, a translation rule is proved sound by showing p and $q^{(n)}$ have the same observable semantics in the symbolic store $\mathcal{U}$ and path condition C, under the assumption that this holds for all p_i, $q^{(n_i)}$ in $\mathcal{U}\circ\mathcal{U}_i$, $C\cup C_i$. A trivial and useless way to prove soundness is by stipulating $obs_i \coloneqq \emptyset$, however, in general we want to show program equivalence for non-trivial *obs*. A *semantic* soundness argument is either informal or else is very hard to automate (based on a formalization of the semantics of both input programming languages). We suggest a practical alternative, based on the assumption that we have sound SE systems for the source and target language at our disposal. The main idea is to

assgnPVTransl

$$\frac{(\mathcal{U} \circ (\mathrm{x} := \mathrm{y}), C, \pi\ \omega \Vdash q^n)@(obs)}{(\mathcal{U}, C, \pi\ \texttt{x=y;}\ \omega \Vdash \left(q \lhd_n \left(\begin{array}{l}\texttt{\%0 = load i32, i32* \%y}\\ \texttt{store i32 \%0, i32* \%x}\end{array}\right)\right)^{(n)})@(obs)}$$

ltCompTransl

$$\frac{(\mathcal{U} \circ (\mathrm{b} := \mathrm{x} < \mathrm{y}), C, \pi\ \omega \Vdash q^n)@(obs)}{(\mathcal{U}, C, \pi\ \texttt{b=x<y;}\ \omega \Vdash \left(q \lhd_n \left(\begin{array}{l}\texttt{\%0 = load i32, i32* \%x}\\ \texttt{\%1 = load i32, i32* \%y}\\ \texttt{\%2 = icmp sle i32 \%0, i32 \%1}\\ \texttt{store i1 \%2, i1* \%b}\end{array}\right)\right)^{(n)})@(obs)}$$

ifElseTransl

$$\frac{\begin{array}{c}(\mathcal{U}, C \cup \{\mathrm{b} \doteq \mathrm{TRUE}\}, \pi\ \widehat{P_1}\ \omega \Vdash (q \lhd_n \widehat{P_2})^{(n)})@(obs_1)\\ (\mathcal{U}, C \cup \{\mathrm{b} \doteq \mathrm{FALSE}\}, \pi\ \widehat{P_1'}\ \omega \Vdash (q \lhd_n \widehat{P_2'})^{(n)})@(obs_2)\end{array}}{(\mathcal{U}, C, \pi\ \begin{array}{l}\texttt{if (b)}\\ \widehat{P_1}\\ \texttt{else}\\ \widehat{P_1'}\end{array}\ \omega \Vdash \left(q \lhd_n \left(\left(\begin{array}{l}\texttt{\%1 = load i1, i1* \%b}\\ \texttt{br i1 \%1, label \%2,}\\ \texttt{\qquad label \%3}\\ \texttt{br label \%4; <label>:\%2}\\ \texttt{br label \%4; <label>:\%3}\\ \texttt{\qquad ; <label>:\%4}\end{array}\right) \lhd_3 \widehat{P_2'} \lhd_2 \widehat{P_2}\right)\right)^{(n)})@(obs_1 \cup obs_2)}$$

Fig. 4: Some example rules of the DSETR

reflect the validity requirement in Def. 5 into our program logic via *justifying formulas* for dual SE states. Thus, the semantic soundness notion is reduced to a DL *formula* which can be proven valid by automated deductive verification.

Definition 7 (Justifying Formula). *Let $s = (\mathcal{U}, C, p \Vdash q^{(n)})@(obs)$ be a dual SE state. For each variable $\mathrm{x} \in obs$, let $p^{\mathrm{Sk}}_{\mathrm{x}}(\mathrm{x})$ be a fresh unary Skolem predicate symbol uniquely associated with* x*. We define the* justifying formula *$F(s)$ of s as*

$$F(s) \coloneqq \{\mathcal{U}\}\left(\bigwedge C\right) \rightarrow \left(\{\mathcal{U}\}[p](obs') \leftrightarrow \{\mathcal{U}\}[q]^{(n)}(obs')\right),$$

where $obs' \coloneqq \bigwedge_{\mathrm{x} \in obs}(p^{\mathrm{Sk}}_{\mathrm{x}}(\mathrm{x}))$.

Proposition 1. *A dual state $s \in \mathbb{S}^{\mathsf{d}}_{SE}$ is valid iff its justifying formula is valid.*

Figure 4 shows rules for variable assignment (the introductory example), less-than comparison, and **if** statement. Consider rule ifElseTransl. If we know how to translate $\widehat{P_1}$ given that the guard b is true, and likewise for $\widehat{P_1'}$, we can define the translation for an *arbitrary* **if** statement.

For practical application of Def. 7 in soundness proofs of a translation rule we have to decide how to handle the abstract parts of the rule that are only instantiated with concrete elements when translating a concrete program. The abstract parts are (1) the context information, i.e. the current symbolic store $\mathcal{U}$

and path condition C, as well as the program contexts $\pi\ \omega$ and q, (2) the sets of observable variables, and (3) abstract program placeholders.

Regarding (1), since $\mathcal{U}$ and C are present in both the conclusion *and* in the premisses of rules, because of the restrictions imposed on translation rules (cf. Fig. 3), we can simply omit them in justifying formulas. Java program contexts are treated as abstract program symbols. For LLVM IR, we introduce an additional simplification technique: Consider the modality $[q \lhd_n q']^{(n+n')}$, where n' is the number of instructions in q'. It can be simplified to $[q]^{(n)}$, since (i) the program q' has already been processed, and (ii) our approach to loop compilation ensures that back jumps exit the modality (see Sect. 6). After this simplification, q in $[q]^{(n)}$ can be handled as an abstract program symbol.

Regarding (2), we over-approximate observable variable sets by introducing fresh constant symbols $p^{\mathrm{Sk}}_{obs_i}(\bar{\mathrm{x}})$ for each symbolic set obs_i, which accept as arguments the program variables $\bar{\mathrm{x}}$ occurring in any state involved in the rule (observable variable sets are generally assumed not to contain anonymous LLVM IR registers). A straightforward representation of a union $obs_1 \cup obs_2$ is the disjunction $p^{\mathrm{Sk}}_{obs_1}(\bar{\mathrm{x}}) \vee p^{\mathrm{Sk}}_{obs_2}(\bar{\mathrm{x}})$. We choose instead the more precise representation of *guarded conjunctions*: Each obs_i is associated to a dual SE state with branch condition C_i. Since our semantic framework for SE stipulates that branch conditions are exhaustive and mutually exclusive, there is always exactly one conjunct of the guarded conjunction $((\bigwedge C_1) \to p^{\mathrm{Sk}}_{obs_1}(\bar{\mathrm{x}})) \wedge ((\bigwedge C_2) \to p^{\mathrm{Sk}}_{obs_2}(\bar{\mathrm{x}}))$ that is valid in a concrete state. Finally, we assume that abstract block contracts are oblivious to anonymous LLVM IR registers, i.e. contract path conditions $C_{\widehat{P}}$ for abstract program symbols do not contain anonymous LLVM IR registers.

Remark 1. Programs p and $q^{(n)}$ might change program variables occurring in path conditions C_i, which is why guarded conjunctions as defined above do not work in general. This issue can be easily addressed by introducing fresh program variables that record the pre-state of all variables in the C_i. To keep the presentation readable, we avoid this technicality by assuming that program variables in path conditions are unchanged.

The subsequent theorem stipulates a generalized notion of justifying formulas instantiated for translation rules, taking into account the above considerations. Essentially, it is a syntactic representation of Def. 6 based on Prop. 1: A translation rule is sound if the conjunction of the generalized justifying formulas of the output states (the rule's premises) implies the formula for the input state (the conclusion). For brevity, we write C instead of $\bigwedge C$ for path conditions.

Theorem 1. *A translation rule with premises $pr_1, \ldots, pr_m$ and conclusion c is sound if the formula $\left(\bigwedge_{i=1,\ldots,m} F'(pr_i)\right) \to F'(c)$ is valid. We define F' as*

$$F'((\mathcal{U} \circ \mathcal{U}', C \cup C', \pi\ p\ \omega \dashv\vdash (q \lhd_n q')^{(n')})@(obs)) \coloneqq \\ \{\mathcal{U}'\}C' \to \left(\{\mathcal{U}'\}[\pi\ p\ \omega](obs') \leftrightarrow \{\mathcal{U}'\}[q \lhd_n q']^{(n')}(obs')\right),$$

where $obs' := p^{\mathrm{Sk}}_{obs}(\bar{\mathtt{x}})$ *if* obs *is a single placeholder; and, if* obs *is a union of symbols* obs_i *arising from premisses* pr_i *with path conditions* C_i, $obs' := \bigwedge_i (C_i \to p^{\mathrm{Sk}}_{obs_i}(\bar{\mathtt{x}}))$. *The predicates* $p^{\mathrm{Sk}}_{obs}(\bar{\mathtt{x}})$ *are fresh, uniquely associated with each* obs, *and* $\bar{\mathtt{x}}$ *are all program variables occurring in* $\mathcal{U}_i$, C_i *of any* pr_i.

Example 3 (Soundness of ifElseTransl*).* To prove rule ifElseTransl sound, by Thm. 1 it suffices to prove the following formula valid (we abbreviate the program in the LLVM IR part of the conclusion with dots):

$$(\mathtt{b} \doteq \mathrm{TRUE} \to ([\pi\ \widehat{P_1}\ \omega](p^{\mathrm{Sk}}_{obs_1}(\mathtt{b})) \leftrightarrow [q \lhd_n \widehat{P_2}]^{(n)}(p^{\mathrm{Sk}}_{obs_1}(\mathtt{b})))) \wedge \tag{1}$$

$$(\mathtt{b} \doteq \mathrm{FALSE} \to ([\pi\ \widehat{P_1'}\ \omega](p^{\mathrm{Sk}}_{obs_2}(\mathtt{b})) \leftrightarrow [q \lhd_n \widehat{P_2'}]^{(n)}(p^{\mathrm{Sk}}_{obs_2}(\mathtt{b})))) \tag{2}$$

$$\to \quad \big(([\pi\ \mathtt{if}\ (\mathtt{b})\ \widehat{P_1}\ \mathtt{else}\ \widehat{P_1'}\ \omega]) \tag{3}$$

$$((\mathtt{b} \doteq \mathrm{TRUE} \to p^{\mathrm{Sk}}_{obs_1}(\mathtt{b})) \wedge (\mathtt{b} \doteq \mathrm{FALSE} \to p^{\mathrm{Sk}}_{obs_2}(\mathtt{b}))) \leftrightarrow$$

$$([q \lhd_n (((\cdots) \lhd_3 \widehat{P_2'}) \lhd_2 \widehat{P_2})]^{(n)} \tag{4}$$

$$((\mathtt{b} \doteq \mathrm{TRUE} \to p^{\mathrm{Sk}}_{obs_1}(\mathtt{b})) \wedge (\mathtt{b} \doteq \mathrm{FALSE} \to p^{\mathrm{Sk}}_{obs_2}(\mathtt{b})))\big)$$

We first define some abbreviations. Let, for $k = 1, 2$,

$$pre^k_1 := \quad C_{\widehat{P_1}} \wedge \{\mathcal{U}^a_{\widehat{P_1}}\} C_{\pi\omega} \to \ \{\mathcal{U}^a_{\widehat{P_1}} \circ \mathcal{U}^a_{\pi\omega}\}(p^{\mathrm{Sk}}_{obs_k}(\mathtt{b}))$$

$$pre^k_2 := \quad C_{\widehat{P_2}} \wedge \{\mathcal{U}^a_{\widehat{P_2}}\} C^a_q \to \quad \{\mathcal{U}^a_{\widehat{P_2}} \circ \mathcal{U}^a_q\}(p^{\mathrm{Sk}}_{obs_k}(\mathtt{b}))$$

and $\overline{pre}^k_i$ similarly for $\widehat{P_i'}$ instead of $\widehat{P_i}$. Treating $\pi\omega$ and q as abstract programs and simplifying $[q \lhd_n \widehat{P_2}]^{(n+1)}$ to $[q]^{(n)}$ as explained above, SE of the modalities in premise (1) by Def. 3 results in pre^1_i, and for (2) in $\overline{pre}^2_i$, $i = 1, 2$. We obtain

$$\mathtt{b} \doteq \mathrm{TRUE} \to (pre^1_1 \leftrightarrow pre^1_2) \quad \text{(1se)}$$

$$\mathtt{b} \doteq \mathrm{FALSE} \to (\overline{pre}^2_1 \leftrightarrow \overline{pre}^2_2) \quad \text{(2se)}$$

The `if` statement in formula (3) causes SE to split. The "then" branch, in which $\widehat{P_1}$ is executed, evaluates to

$$\mathtt{b} \doteq \mathrm{TRUE} \to \big(C_{\widehat{P_1}} \wedge \{\mathcal{U}^a_{\widehat{P_1}}\} C_{\pi\omega} \to \big(\{\mathcal{U}^a_{\widehat{P_1}} \circ \mathcal{U}^a_{\pi\omega}\}((\mathtt{b} \doteq \mathrm{TRUE} \to p^{\mathrm{Sk}}_{obs_1}(\mathtt{b})) \wedge (\mathtt{b} \doteq \mathrm{FALSE} \to p^{\mathrm{Sk}}_{obs_2}(\mathtt{b})))\big)\big)$$

Due to the premise $\mathtt{b} \doteq \mathrm{TRUE}$, the right conjunct of the guarded conjunction simplifies to true and can be removed, and similarly for the "else" branch of the `if` statement. Therefore, formula (3) simplifies to

$$\mathtt{b} \doteq \mathrm{TRUE} \to pre^1_1 \wedge \mathtt{b} \doteq \mathrm{FALSE} \to \overline{pre}^2_1 \quad \text{(3se)}$$

SE of (4) similarly splits and produces two conjuncts. Again, we focus on the "then" branch where the dots abbreviate the guarded conjunctions of $p^{\mathrm{Sk}}_{obs_i}(\mathtt{b})$:

$$\big(\{\%1 := \mathtt{b}\}(\%1 \doteq \mathrm{TRUE}) \to \big(\{\%1 := \mathtt{b}\} C_{\widehat{P_2}} \wedge \{(\%1 := \mathtt{b}) \circ \mathcal{U}^a_{\widehat{P_2}}\} C^a_q \to \{(\%1 := \mathtt{b}) \circ \mathcal{U}^a_{\widehat{P_2}} \circ \mathcal{U}^a_q\}(\cdots)\big)\big)$$

$$\textsf{llvmLoopScopeEnter}\ \frac{(\mathcal{U} \circ \mathcal{U}_{havoc}, C, (q \ \lhd_n \ \circlearrowright_{\mathrm{x}})^{(n+1)})}{(\mathcal{U}, C, (q \ \lhd_n \ \circlearrowright_{\mathrm{x}})^{(n)})} \qquad \textsf{llvmLoopScopeExit}\ \frac{(\mathcal{U} \circ (\mathrm{x} \doteq \mathrm{TRUE}), C, (q \ \lhd_n \ {}_{\mathrm{x}}\circlearrowleft)^{(n+1)})}{(\mathcal{U}, C, (q \ \lhd_n \ {}_{\mathrm{x}}\circlearrowleft)^{(n)})}$$

$$\textsf{llvmLoopScopeCont}\ \frac{(\mathcal{U} \circ (\mathrm{x} := \mathrm{FALSE}), C, \_)}{(\mathcal{U}, C, \left(q \lhd_n \left(\begin{pmatrix} \circlearrowright_{\mathrm{x}} \\ \texttt{br label \%}i \\ \qquad\qquad \texttt{; label \%i} \\ \texttt{br label \%}j \\ {}_{\mathrm{x}}\circlearrowleft \end{pmatrix} \begin{matrix} \lhd_3 \ q_2 \\ \lhd_2 \ q_1 \end{matrix} \right)^{(n')} \right))} \ (*)$$

where $(*)$ n' points to the position of the second `br` statement, and $\%i$ is the label for the basic block indicated by the comment (starting after the first `br`).

Fig. 5: LLVM IR loop scope rules

The crucial difference to the Java part consists in the update $\%1 := \mathtt{b}$ resulting from loading the value of b into the anonymous register %1. We have to perform a further simplification step. Generally, we can drop an update $\mathcal{U}_1$ in a formula $\{\mathcal{U}_1 \circ \mathcal{U}_2\}\varphi$ if φ does not contain any left-hand side of $\mathcal{U}_1$, and $\mathcal{U}_2$ does not contain as right-hand side any left-hand side of $\mathcal{U}_1$ [1, Chapter 3]. This simplification is applicable here, because we assumed abstract block contracts to be ignorant about anonymous LLVM IR registers: Formula $\{(\%1 := \mathtt{b}) \circ \mathcal{U}^a_{\widehat{P_2}}\}C^a_q$ is simplified to $\{\mathcal{U}^a_{\widehat{P_2}}\}C^a_q$, because %1 does not appear in $C_{\widehat{P_2}}$ and not as a right-hand side in $\mathcal{U}^a_{\widehat{P_2}}$, which only assigns fresh constants (similarly for $\{\%1 := \mathtt{b}\}C_{\widehat{P_2}}$). Formula $\{(\%1 := \mathtt{b}) \circ \mathcal{U}^a_{\widehat{P_2}} \circ \mathcal{U}^a_q\}(\dots)$ is simplified to $\{\mathcal{U}^a_{\widehat{P_2}} \circ \mathcal{U}^a_q\}(\dots)$, because $\mathcal{U}^a_{\widehat{P_2}}, \mathcal{U}^a_q$ do not have %1 as right-hand side. Finally, $\{\%1 := \mathtt{b}\}\,(\%1 \doteq \mathrm{TRUE})$ simplifies to $\mathtt{b} := \mathrm{TRUE}$. After simplification of the guarded predicates as before, we obtain

$$\mathtt{b} \doteq \mathrm{TRUE} \to pre^1_2 \ \wedge \mathtt{b} \doteq \mathrm{FALSE} \to \overline{pre}^2_2 \quad \text{(4se)}$$

In summary, we have reduced soundness of ifElseTransl to the following propositional formula, which is a tautology:

$$\begin{aligned} &\big(\big(\mathtt{b} \doteq \mathrm{TRUE} \to (pre^1_1 \leftrightarrow pre^1_2)\big) \ \wedge && \text{(1se)}\\ &\big(\mathtt{b} \doteq \mathrm{FALSE} \to (\overline{pre}^2_1 \leftrightarrow \overline{pre}^2_2)\big)\big) \quad \to && \text{(2se)}\\ &\big(\big(\mathtt{b} \doteq \mathrm{TRUE} \to pre^1_1 \wedge \mathtt{b} \doteq \mathrm{FALSE} \to \overline{pre}^2_1\big) \ \leftrightarrow && \text{(3se)}\\ &\big(\mathtt{b} \doteq \mathrm{TRUE} \to pre^1_2 \ \wedge \mathtt{b} \doteq \mathrm{FALSE} \to \overline{pre}^2_2\big)\big) && \text{(4se)} \end{aligned}$$

□

The soundness proof just sketched is well in the realm of what is automatically provable with a deductive verification system such as KeY [1].

6 Handling of Loops

Symbolic execution of LLVM IR branch instructions and compilation of loops in a manner that ensures termination poses a challenge: Since LLVM IR pro-

grams may contain *back*- and forward jumps, SE of branching rules such as IlvmCondBrUnrestr (Fig. 2) might diverge. The standard approach to ensure termination in *structured* programming languages uses *loop invariant* rules. Our solution for the unstructured case is based on the more general concept of *loop scopes*, introduced in [17] for Java. A (Java) loop scope statement $\circlearrowleft_{\mathtt{x}}\, p\, {}_{\mathtt{x}}\circlearrowleft$ defines a scope for its body p; it has a boolean *index variable* x which is passed to the delimiters $\circlearrowleft_{\mathtt{x}}$ and ${}_{\mathtt{x}}\circlearrowleft$ as a parameter. One can read $\circlearrowleft_{\mathtt{x}}\, p\, {}_{\mathtt{x}}\circlearrowleft$ as "execute loop body p in its own scope with continuation information x". Loops are transformed into loop scope statements. Whenever a loop normally would execute a further iteration, SE of the loop scope terminates, and x is set to FALSE. When the loop would be exited, symbolic execution continues after the loop scope, and x is set to TRUE.

We implement this idea for LLVM IR: First, existing rules for jumps, such as IlvmCondBreakUnrestr, are restricted to *forward* jumps. Second, we add new rules (Fig. 5) for loop scopes, including one for backward jumps. As there are no explicit loops in LLVM IR, we insert scope delimiters $\circlearrowleft_{\mathtt{x}}$, ${}_{\mathtt{x}}\circlearrowleft$ at suitable positions. We assume LLVM IR programs correspond to well-formed Java loops.

Rule IlvmLoopScopeEnter begins symbolic execution of a loop scope by setting all variables assigned in the scope to fresh constants via an update $\mathcal{U}_{havoc}$, because we cannot know their value during an arbitrary iteration. (In Java, this task is performed by the loop invariant rule.) Rule IlvmLoopScopeExit signals that the loop scope has been exited by setting the index variable to TRUE. The only rule applicable to backward jumps is IlvmLoopScopeCont. Using the loop scope concept constitutes a restriction of the LLVM IR fragment on which SE is complete; however, that fragment is sufficient for our compilation target.

Rule whileTransl in Fig. 6 is a translation rule for **`while`** loops based on loop scopes. In the LLVM IR code, "only" the instruction pointer is incremented, corresponding to an execution of the opening loop scope statement. Its structure corresponds closely to the Java part in the premise. It includes a **`br`** instruction (labelled %5) which is dead due to the translation of the **`break`** before. It is included to match the output of the rule ifElseTransl (Fig. 4—the removal of dead basic blocks is the task of subsequent simplification steps).

To execute a dual SE state with a loop, we apply whileTransl and ifElseTransl, execute both **`if`** branches, and finally apply continueLoopScopeTransl as well as breakLoopScopeTransl. We keep the LLVM IR loop scopes in the conclusion of whileTransl for handling the backward jump in the last **`br`** instruction. In a post-processing step, loop scopes can safely be dropped: If a program is correct in the restricting presence of loop scopes, it is also sound without. An example for the translation of a factorial method, which contains a loop, is given in the appendix.

7 Related Work

Our work takes up the concept of "dual" SE states and SE-based compilation introduced in [8] (see also [9]), where compilation from Java to Bytecode was

whileTransl

$$\frac{(\mathcal{U}\circ\mathcal{U}_{havoc}, C, \pi \begin{pmatrix} \circlearrowleft_{\mathrm{x}}\texttt{if (}b\texttt{) \{} \\ \widehat{P_1} \\ \texttt{continue;} \\ \texttt{\} else \{} \\ \texttt{break;} \\ \texttt{\}}\ {}_{\mathrm{x}}\circlearrowleft \end{pmatrix} \omega \dashv\vdash \ldots^{(n+2)})@(obs)}{(\mathcal{U}, C, \pi \ \overset{\texttt{while (}b\texttt{)}}{\widehat{P_1}}\ \omega \dashv\vdash \left(q \lhd_n \left(\left(\begin{array}{ll} \circlearrowleft_{\mathrm{x}} & \\ \texttt{br label \%1} & \\ & \texttt{; <label>:\%1} \\ \texttt{\%2 = load i1, i1* \%}b & \\ \texttt{br i1 \%2, label \%3,} & \\ \texttt{label \%4} & \\ \texttt{br label \%6} & \texttt{; <label>:\%3} \\ \texttt{br label \%7} & \texttt{; <label>:\%4} \\ \texttt{br label \%6} & \texttt{; <label>:\%5} \\ \texttt{br label \%1} & \texttt{; <label>:\%6} \\ {}_{\mathrm{x}}\circlearrowleft & \texttt{; <label>:\%7} \end{array} \right) \lhd_4 \widehat{P_1} \right)^{(n)} \right))@(obs)}$$

where $\mathcal{U}_{havoc}$ is as above. The dots in the premise indicate that LLVM IR instructions there are the same as in the conclusion.

breakLoopScopeTransl

$$\frac{(\mathcal{U}\circ(\mathrm{x} := \mathrm{TRUE}), C, \pi\ \omega \dashv\vdash q^{(n)})@(obs)}{(\mathcal{U}, C, \pi\ \circlearrowleft_{\mathrm{x}}\texttt{break;}\ {}_{\mathrm{x}}\circlearrowleft\ \omega \dashv\vdash \left(q \lhd_n \left(\begin{array}{ll} \circlearrowleft_{\mathrm{x}} & \\ \texttt{br label \%1} & \\ \texttt{br label \%3} & \texttt{; <label>:\%1} \\ \texttt{br label \%1} & \texttt{; <label>:\%2} \\ {}_{\mathrm{x}}\circlearrowleft & \texttt{; <label>:\%3} \end{array} \right) \right)^{(n+2)})@(obs)}$$

continueLoopScopeTransl

$$\frac{(\mathcal{U}\circ(\mathrm{x} := \mathrm{FALSE}), C, \_ \dashv\vdash \_)@(obs)}{(\mathcal{U}, C, \pi\ \circlearrowleft_{\mathrm{x}}\texttt{continue;}\ {}_{\mathrm{x}}\circlearrowleft\ \omega \dashv\vdash \left(q \lhd_n \left(\begin{array}{ll} \circlearrowleft_{\mathrm{x}} & \\ \texttt{br label \%1} & \\ \texttt{br label \%1} & \texttt{; <label>:\%1} \\ {}_{\mathrm{x}}\circlearrowleft & \end{array} \right) \right)^{(n+2)})@(obs)}$$

Fig. 6: Translation rules for loops

studied as an application of a program transformation framework. The system proposed there is not based on a formalization of the target language, therefore, the employed correctness notions remain underspecified. We define correctness differently, based on a new semantics for symbolic execution (not depending on a validity calculus) and focus on how to show the correctness of translation rules automatically. For the execution of abstract programs, we generalize abstract operation contracts [4, 7] to abstract block contracts. In contrast to the former, where contracts are abstract, but programs are concrete, our blocks may consist of abstract program symbols. This gives rise to the concept of *abstract execution* that permits to reduce a limited second-order inference (no induction, no higher-order quantification) about programs to first-order dynamic logic. The resulting formulas can be proven automatically by a program verifier.

The two most relevant research areas to compare with are: (1) program logics, formal models, and symbolic execution for low-level languages, in particular LLVM IR, and (2) rule-based or certified correct compilation and program transformation. Vellvm [19], implemented in Coq, provides an operational semantics of LLVM IR based on a formalized memory model. The authors name as long-term goal to provide a semantic basis for a fully verified LLVM IR compiler. Our scope is verification of correctness and behavioral equivalence, i.e. of functional and relational properties, which is reflected in our more high-level formalization. In future work, it would be interesting to investigate whether it can be proved with Vellvm that our SE rules faithfully model their LLVM IR semantics. The only (implemented) SE system for LLVM IR we know of is KLEE [6], an automatic tool, mainly aiming at creating high-coverage test suites and unveiling generic errors, such as memory faults. Various program logics for low-level, unstructured programs have been suggested in the literature, for example, in [15]. This results in theoretically complete proof systems, however, it is difficult to come up with suitable intermediate assertions during verification. Since we target *compilation* and not functional verification, it suffices to use schematic intermediate assertions that are built into our rules. We envision an implementation of our SE framework for LLVM IR in a logic-based system like KeY [1], facilitating semi-interactive proofs of complex functional properties of LLVM IR code.

Regarding area (2), Rotan [3] is a rule-based compiler for parallel programs. It features a specialized rule language and puts focus on modularity. We currently only consider sequential programs, but achieve fully automated correctness proofs of our rules. Of great interest is the work on verified compilers. A prominent example is the CompCert project [13], a verified compiler from C to PowerPC assemblies (mostly) written in Coq. CompCert covers all compilation phases including optimization. Its correctness notion, "forward simulation for safe programs", is more flexible than, e.g., bisimulation. Our approach only takes the *outcome* of source and target programs (its big-step semantics) into account, and not *how* it is computed (its small-step semantics). Therefore, our justifying formulas permit more far-reaching optimizations, abstracting away from concrete control flow. To some degree, CompCert follows different goals than our approach. Its extensive formalizations permit meta proofs about pro-

grams (e.g., taking into account a realistic memory model), while we only formalize operational semantics and relate source and target. A strength of our approach is its modularity, entailed by its rule-based nature; furthermore, Coq proofs are interactive, whereas our goal is to automate the proofs of the translation rules. A related and noteworthy system which shares our source language Java is Jinja [10], a formalization in Isabelle/HOL of semantics, virtual machine and compiler of a Java-like language including a mechanized proof that the compiler preserves the semantics. This system is also based on an interactive proof assistant and requires manual proof effort. CakeML [18] is a more recent verified compiler for a functional programming language developed in Isabelle/HOL which invests a lot of effort in modularity; still, the backend does not provide automated support for correctness proofs of individual source elements.

For parts of the compilation process, automated solutions exist: One example is Alive [14], a verification/code generation framework for peephole optimizations in LLVM. Within Alive, it is possible to formalize local algebraic simplifications and code optimizations in a restricted DSL, which are then transformed to first-order logic assertions that are passed to an SMT solver. While being automatic, the addressed problem is relatively simple compared to full-fledged compilation.

8 Conclusion and Future Work

We presented a theoretical framework for correct-by-construction, modular rule-based compilation from Java to LLVM IR. The framework is based on a new semantics of Symbolic Execution (SE) and a formalization of the source and target languages in terms of SE rules, which constitutes the trust anchor. For the Java part, we build on Java DL, the mature logical basis of the KeY system [1]. Our new (partial) SE system for LLVM IR is designed to integrate smoothly with it. Translation rules are proven correct automatically by reducing correctness assertions about abstract programs to first-order dynamic logic formulas, which can be fed to an automatic program verifier.

There are several interesting directions for future work which we plan to pursue: We aim to extend an existing formalization of our SE semantics in Coq to our compilation framework, instantiated to a structured, imperative and a simple unstructured language. Our current main focus is the integration of the SE system for at least a subset of LLVM IR into the Java verifier KeY. Based on this, we will construct a compiler for sequential Java realizing the theoretical framework outlined in this paper. Since an SE engine that simplifies away all complex Java constructs and that produces SSA is available in KeY already [1, Chapter 3], this is likely to succeed quickly. KeY's program logic can also deal with complex Java constructs such as method calls, for example, using method frames and non-schematic program transformation rules. Again, these techniques are transferable to LLVM IR compilation. We intend to add a rule-based modular optimization phase, allowing for high-level optimizations exploiting the global knowledge gained by Symbolic Execution. Our correctness notion, which is more general than (bi-)simulation used elsewhere, because it

takes the context into account and permits abstract programs, should be adequate for this purpose. Finally, it would be natural to attach our framework as a backend to a refinement-based correct-by-construction tool. This opens the possibility of an "end-to-end" correctness-by-construction approach—from declarative specifications to executable machine code.

References

1. Ahrendt, W., Beckert, B., et al. (eds.): Deductive Software Verification – The KeY Book, LNCS, vol. 10001. Springer (2016)
2. Augustsson, L.: A Compiler for Lazy ML. In: Proc. LFP'84. ACM (1984)
3. Breebaart, L.: Rule-based Compilation of Data Parallel Programs. Ph.D. thesis, Delft University of Technology (2003)
4. Bubel, R., Hähnle, R., et al.: Fully Abstract Operation Contracts. In: 6th Intern. Symp. of Leveraging Applications of FM, ISoLA 2014. pp. 120–134 (2014)
5. Burstall, R.M.: Program Proving as Hand Simulation with a Little Induction. In: Information Processing, pp. 308–312. Elsevier (1974)
6. Cadar, C., Dunbar, D., et al.: KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In: 8th USENIX Conference on OSDI. pp. 209–224. USENIX Association, Berkeley, CA, USA (2008)
7. Hähnle, R., Schaefer, I., et al.: Reuse in Software Verification by Abstract Method Calls. In: Proc. 24th Intern. Conf. on Automated Deduction. pp. 300–314 (2013)
8. Ji, R.: Sound Program Transformation Based on Symbolic Execution and Deduction. Ph.D. thesis, Technische Universität Darmstadt (2014)
9. Ji, R., Bubel, R., et al.: Program transformation and compilation. In: Ahrendt, W., Beckert, B., et al. (eds.) Deductive Software Verification—The KeY Book: From Theory to Practice, LNCS, vol. 10001, chap. 14, pp. 473–492. Springer (2016)
10. Klein, G., Nipkow, T.: A Machine-Checked Model for a Java-like Language, Virtual Machine, and Compiler. ACM Trans. PLS 28(4), 619–695 (Jul 2006)
11. Lattner, C., Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: Proc. CGO'04. p. 75. IEEE Comp. Soc. (2004)
12. Le, V., Afshari, M., et al.: Compiler Validation via Equivalence Modulo Inputs. In: Proc. 35th ACM SIGPLAN Conf. on PLDI. pp. 216–226. ACM (2014)
13. Leroy, X.: Formal Verification of a Realistic Compiler. Communications of the ACM 52(7), 107–115 (2009)
14. Menendez, D., Nagarakatte, S., et al.: Alive-FP: Automated Verification of Floating Point Based Peephole Optimizations in LLVM. In: Rival, X. (ed.) Proc. Intern. Static Analysis Symp. pp. 317–337. Springer (2016)
15. Saabas, A., Uustalu, T.: A compositional natural semantics and Hoare logic for low-level languages. Theoretical Computer Science 373(3), 273–302 (2007)
16. Scheurer, D., Hähnle, R., et al.: A General Lattice Model for Merging Symbolic Execution Branches. In: Ogata, K., Lawford, M., et al. (eds.) Proc. of 18th Intern. Conf. on Formal Engineering Methods. pp. 57–73. Springer (2016)
17. Steinhöfel, D., Wasser, N.: A New Invariant Rule for the Analysis of Loops with Non-standard Control Flows. In: Polikarpova, N., Schneider, S. (eds.) Proc. of 13th Intern. Conf. on Integrated Formal Methods. pp. 279–294. Springer (2017)
18. Tan, Y.K., Myreen, M.O., et al.: A New Verified Compiler Backend for CakeML. In: Proc. 21st Intern. Conf. on Functional Programming. pp. 60–73. ACM (2016)
19. Zhao, J., Nagarakatte, S., et al.: Formalizing the LLVM Intermediate Representation for Verified Program Transformations. In: Proc. 39th ACM SIGPLAN-SIGACT Symp. on POPL. pp. 427–440. ACM (2012)

Appendix

This appendix provides additional material for the reader, including a sketch of the integration of state merging techniques into our approach, an extended compilation example, and a proof of Thm. 1.

A State Merging

Our SE transition relation defined in Sect. 3 outputs tree structures: all states, but the root, have exactly one predecessor. *State Merging* [16] has originally been invented to mitigate the state explosion problem of SE. It permits to merge symbolic states with identical program counter as arising, for example, after the execution of the then and else branch of an `if` statement, thus reducing the state space. Def. 2 can be extended to transition relations with state merging by applying the same conditions reversely for "merging" transitions:

$$\textsf{mergeTransI}\quad \frac{(\mathcal{U}_{merge}, \bigwedge C_1 \vee \bigwedge C_2, p \dashv\Vdash q^{(n)})@(obs_1 \cup obs_2)}{\begin{array}{c}(\mathcal{U}_1, C_1, p \dashv\Vdash q^{(n)})@(obs_1)\\ (\mathcal{U}_2, C_2, p \dashv\Vdash q^{(n)})@(obs_2)\end{array}}$$

where $\mathcal{U}_{merge}$ is a merging update.

Fig. 7: State merging translation rule

Definition 8 (Soundness of SE with State Merging). *Let δ_m be an SE transition relation with state merging (some states have more than one predecessor). We call δ_m* sound *iff the transition relation*

$$\begin{aligned}\delta' \coloneqq &\{(i,o) \in \delta_m : \neg\exists i' \neq i, (i',o) \in \delta_m\} \cup \\ &\{(o,i) : (i,o) \in \delta_m \wedge \exists i' \neq i, (i',o) \in \delta_m\}\end{aligned}$$

is sound in the sense of Def. 2.

Translation rule mergeTransI (Fig. 7) merges two branches with identical program counters. Without it, compiling programs with branching statements leads to duplicated code, for example, in rule ifElseTransI. The double horizontal line below the rule's premise indicates its special status: it is the only rule with more than one conclusion. For constructing the merged symbolic store, there is more than one option [16]. In the simplest case, we create a fully precise value summary by means of an if-then-else term.

B Factorial Example

The program in Fig. 8 (p. 22) computes the factorial of variable `x`, contained in variable `res` after termination. Figure 12 shows the SET of the dual SE states

corresponding to the translation of the Java to LLVM IR code. Constructing the tree works by first symbolically executing the Java program. The Java SET has the same structure, without the LLVM IR parts and observable variables sets.

In the second phase, we apply translation rules from the leaves to the root, obtaining the dual SE states. The root of the tree contains the compiled LLVM IR program q_{result} depicted in Fig. 9 (with dropped loop scopes). The tree is slightly simplified: we sometimes abbreviate programs with "..." and combine several translation rules into one, for example, in the translation of Java's `i++`, which in the course of SE would be simplified to SSA form first. Two subtrees have been factored out for better readability (Figures 10 and 11).

The example also shows an artifact occurring due to our rule-based compilation: The basic block with label `%13` is never reached since it is not targeted by jumps, and the block before jumps outside the loop (this is the translation of the `break` instruction). The instruction has been added because of the general definition of the `if` translation rule; if the else block did not jump out of the loop, it would also make sense. Removal of such dead code, as well as other optimizations, are subject to subsequent post-processing steps and not within the scope of this paper.

C Proofs

Proof (Thm. 1). We must prove that a translation rule is sound (Def. 6) if $\mathcal{F} = \big(\bigwedge_{i=1,\ldots,m} F'(pr_i)\big) \to F'(c)$ is valid. By Prop. 1, a dual SE state is valid iff its justifying formula is valid, and the shape of $\mathcal{F}$ directly encodes Def. 6. Hence, it suffices to show that the validity of justifying formulas $F'(s')$ for *abstract* states s' implies the validity of $F(s)$ for all *concrete instantiations* s of s'. Let $s' = (\mathcal{U}^a \circ \mathcal{U}, C^a \cup C, p_j^{abs} \dashv\vdash p_{IR}^{abs})@(obs^a)$ be an abstract SE state where $\mathcal{U}^a$, C^a and obs^a are symbolic and p_j^{abs}, p_{IR}^{abs} contain abstract placeholder symbols. An instantiation s has the shape $(\mathcal{U}' \circ \mathcal{U}, C' \cup C, p_j \dashv\vdash p_{IR})@(obs)$, where p_j and p_{IR} result from p_j^{abs} and p_{IR}^{abs} by replacing placeholders with concrete programs. We have to show the following implication:

$$\begin{aligned} &\models (\{\mathcal{U}\}C \to (\{\mathcal{U}\}[p_j^{abs}]obs'_{F'} \leftrightarrow \{\mathcal{U}\}[p_{IR}^{abs}]obs'_{F'})) \qquad \Longrightarrow \\ \models (\{\mathcal{U}' \circ \mathcal{U}\}(C' \cup C) &\to (\{\mathcal{U}' \circ \mathcal{U}\}[p_j]obs'_F \leftrightarrow \{\mathcal{U}' \circ \mathcal{U}\}[p_{IR}]obs'_F)) \end{aligned}$$

Since the validity of $\{\mathcal{U}' \circ \mathcal{U}\}(C' \cup C)$ implies the validity of $\{\mathcal{U}\}C$, it suffices to show the following, stronger property:

$$\begin{aligned} &\models ((\{\mathcal{U}\}[p_j^{abs}]obs'_{F'} \leftrightarrow \{\mathcal{U}\}[p_{IR}^{abs}]obs'_{F'})) \implies \\ &\models ((\{\mathcal{U}' \circ \mathcal{U}\}[p_j]obs'_F \leftrightarrow \{\mathcal{U}' \circ \mathcal{U}\}[p_{IR}]obs'_F)) \end{aligned} \tag{*}$$

Since abstract execution is sound (Obs. 1), the premise of (*) holds for all concrete contracts substituted for the abstract ones induced by the placeholders in the programs; this means that in particular, we can substitute p_j/p_{IR} for p_j^{abs}/p_{IR}^{abs}. Let $\mathcal{U}_{Java}, \mathcal{U}_{IR}$ be the updates resulting from SE of the concrete programs (if SE splits, we can obtain summaries by state merging, see Appendix A).

Now there are two possibilities: Either, $obs'_{F'}$ is an atom, or it is a guarded conjunction (and s constitutes the conclusion of a dual SE rule application). We focus on the more complicated second case. Let obs be the (concrete) set of observable variables of s, C_i the path conditions of the premisses of the rule that has s in its conclusion, and $\bar{\mathrm{x}}$ the variables occurring in any symbolic store or path condition of those premises. Then, after instantiation, (*) expands to:

$$\models \big(\{\mathcal{U}\circ\mathcal{U}_{Java}\}(\bigwedge_i (C_i \to p^{\mathrm{Sk}}_{obs_i}(\bar{\mathrm{x}}))) \leftrightarrow \{\mathcal{U}\circ\mathcal{U}_{IR}\}(\bigwedge_i (C_i \to p^{\mathrm{Sk}}_{obs_i}(\bar{\mathrm{x}})))\big) \implies$$
$$\models \big(\{\mathcal{U}'\circ\mathcal{U}\circ\mathcal{U}_{Java}\}(\bigwedge_{\mathrm{x}\in obs} p^{\mathrm{Sk}}_{\mathrm{x}}(\mathrm{x})) \leftrightarrow \{\mathcal{U}'\circ\mathcal{U}\circ\mathcal{U}_{IR}\}(\bigwedge_{\mathrm{x}\in obs} p^{\mathrm{Sk}}_{\mathrm{x}}(\mathrm{x}))\big)$$

Generally, $\bar{\mathrm{x}}$ can contain variables not occurring in obs, due to over-approximation. On the other hand, if obs contains variables not in $\bar{\mathrm{x}}$, those are not in the scope of the currently investigated rule, and are therefore taken out of consideration here.[2] Additionally, we perform a strengthening by dropping the updates $\mathcal{U}'$ in the conclusion (this can be regarded as an abstraction step, since we discharge information). W.l.o.g., consider two different variables x, y and $obs = \{\mathrm{x}\}$; their right-hand sides in the updates $\mathcal{U}\circ\mathcal{U}_{Java}$ and $\mathcal{U}\circ\mathcal{U}_{IR}$ are t^{x}_{Java}, t^{y}_{IR}, etc. Under the assumption that x and y do not occur in the C_i (see Remark 1), the problem simplifies to:

$$\models \big(\bigwedge_i (C_i \to p^{\mathrm{Sk}}_{obs_i}(t^{\mathrm{x}}_{Java}, t^{\mathrm{y}}_{Java})) \leftrightarrow \bigwedge_i (C_i \to p^{\mathrm{Sk}}_{obs_i}(t^{\mathrm{x}}_{IR}, t^{\mathrm{y}}_{IR}))\big) \implies \qquad (\dagger)$$
$$\models \big(p^{\mathrm{Sk}}_{\mathrm{x}}(t^{\mathrm{x}}_{Java}) \leftrightarrow p^{\mathrm{Sk}}_{\mathrm{x}}(t^{\mathrm{x}}_{IR})\big) \qquad (\triangle)$$

For simplicity, we assume that the terms do not contain program variables. The equivalence in $(\triangle)$ is valid iff for all interpretations $\mathcal{I}$, it holds that

$$\mathcal{I}(t^{\mathrm{x}}_{Java}) \in \mathcal{I}(p^{\mathrm{Sk}}_{\mathrm{x}}) \iff \mathcal{I}(t^{\mathrm{x}}_{IR}) \in \mathcal{I}(p^{\mathrm{Sk}}_{\mathrm{x}}) \qquad (\triangle^{sem})$$

Let $\mathcal{I}_0$ be an arbitrary interpretation; we show that $(\triangle^{sem})$ holds for $\mathcal{I}_0$. All C_i are mutually exclusive due to the restriction built into our semantics (Def. 2), i.e. $C_i \leftrightarrow \bigwedge_{k\neq i} \neg C_k$ for all $i = 1, \ldots, n$. Since $(\dagger)$ is valid, for *any* interpretation $\mathcal{I}$ there is *exactly one* i such that

$$(\mathcal{I}(t^{\mathrm{x}}_{Java}), \mathcal{I}(t^{\mathrm{y}}_{Java})) \in \mathcal{I}(p^{\mathrm{Sk}}_{obs_i}) \iff (\mathcal{I}(t^{\mathrm{x}}_{IR}), \mathcal{I}(t^{\mathrm{y}}_{IR})) \in \mathcal{I}(p^{\mathrm{Sk}}_{obs_i}) \qquad (\dagger^{sem})$$

We choose an interpretation $\mathcal{I}_1$ that (i) interprets $t^{\mathrm{x}}_{Java}, t^{\mathrm{x}}_{IR}$ in the same way as $\mathcal{I}_0$, (ii) satisfies $\mathcal{I}_1(p^{\mathrm{Sk}}_{obs_i}) = \{(d_1, d_2) : d_1 \in \mathcal{I}_0(p^{\mathrm{Sk}}_{\mathrm{x}})\}$ and, at the same time, (iii) satisfies $(\dagger^{sem})$ for some i. By definition of $\mathcal{I}_1$ we have that $(\mathcal{I}_1(t^{\mathrm{x}}_{Java}), \mathcal{I}_1(t^{\mathrm{y}}_{Java})) \in \mathcal{I}_1(p^{\mathrm{Sk}}_{obs_i})$ implies $\mathcal{I}_0(t^{\mathrm{x}}_{Java}) \in \mathcal{I}_0(p^{\mathrm{Sk}}_{\mathrm{x}})$, and, similarly, for t^{x}_{IR}, t^{y}_{IR}. Hence, $(\triangle^{sem})$ holds in $\mathcal{I}_0$. □

[2] When using Thm. 1 to automatically prove the soundness of translation rules with a deductive verification system such as KeY, this aspect can and must be precisely modeled. This can be done by tracking for each abstract program an (abstract) set of variables it depends on.

```
int i=1;

while (
    i < x

  ) {
  res = res*i;

  i++;

}
```

Fig. 8: Factorial program in Java

```
%i = alloca i32
store i32 1, i32* %i
br label %1
%b = alloca i1 ; <label>:%1
%2 = load i32, i32* %i
%3 = load i32, i32* %x
%4 = icmp sle i32 %2, %3
store i1 %4, i1* %b
%5 = load i1, i1* %b
br i1 %5, label %6, label %12
                ; <label>:%6
%7 = load i32, i32* %res
%8 = load i32, i32* %i
%9 = mul i32 %7, %8
store i32 %9, i32* %res
%10 = load i32, i32* %i
%11 = add i32 %11, 1
store i32 %11, i32* %i
br label %14
br label %15   ; <label>:%12
br label %14   ; <label>:%13
br label %1    ; <label>:%14
               ; <label>:%15
```

Fig. 9: Factorial program in LLVM IR

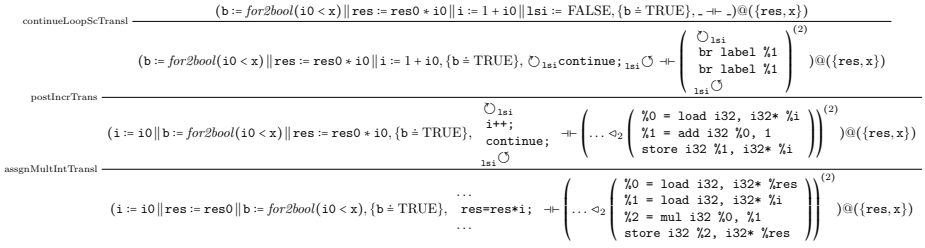

Fig. 10: Subtree "then" for the example tree in Figure 12

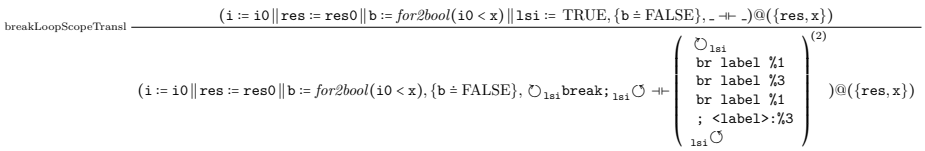

Fig. 11: Subtree "else" for the example tree in Figure 12

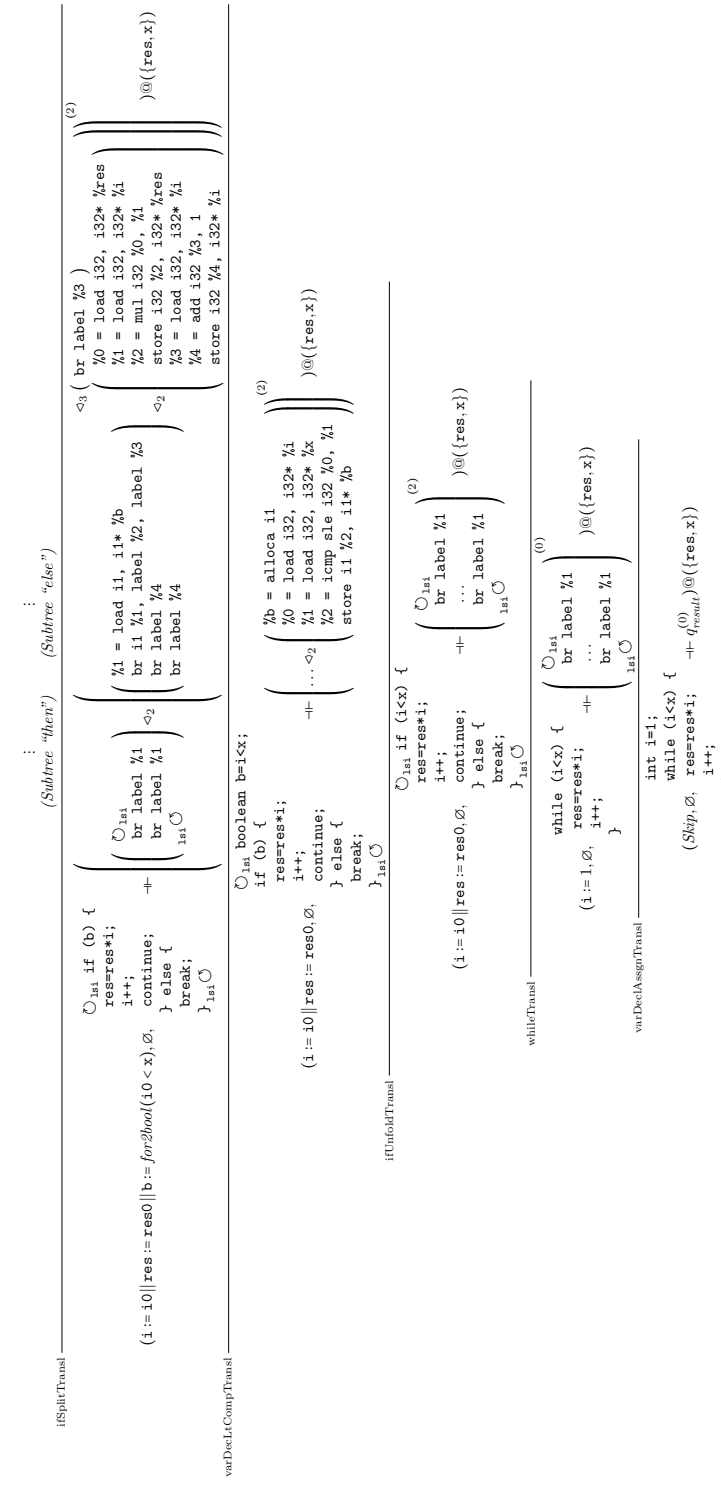

Fig. 12: An example compilation tree (slightly simplified)