An Operational Semantics for Constraint-Logic Imperative Programming

Jan C. Dageförde(✉) and Herbert Kuchen

ERCIS, University of Münster, Münster, Germany
{dagefoerde,kuchen}@uni-muenster.de

Abstract. Object-oriented (OO) languages such as Java are the dominating programming languages nowadays, among other reasons due to their ability to encapsulate data and operations working on them, as well as due to their support of inheritance. However, in contrast to constraint-logic languages, they are not particularly suited for solving search problems. During development of enterprise software, which occasionally requires some search, one option is to produce components in different languages and let them communicate. However, this can be clumsy.

As a remedy, we have developed the constraint-logic OO language Muli, which augments Java with logic variables and encapsulated search. Its implementation is based on a symbolic Java virtual machine that supports constraint solving and backtracking. In the present paper, we focus on the non-deterministic operational semantics of an imperative core language.

Keywords: Java · Operational semantics · Encapsulated search
Programming paradigm integration

1 Introduction

Contemporary software development is dominated by object-oriented (OO) programming. Its programming style benefits most industry applications by providing e.g. inheritance and encapsulation of structure and behaviour, since these concepts can positively contribute towards reusability and maintainability [13]. Nevertheless, some industry applications require search, for which constraint-logic programming is more suited than OO (or imperative) programming. However, developing applications that integrate both worlds, e.g. a Java application using a Prolog search component via Java Native Interface (JNI), is tedious and error-prone [10].

For that reason, we propose the *Münster Logic-Imperative Programming Language (Muli)*, integrating constraint-logic programming with OO programming in a novel way. Based on Java, it adds logic variables and encapsulated search to

D. Seipel et al. (Eds.): DECLARE 2017, LNAI 10997, pp. 64–80, 2018.
https://doi.org/10.1007/978-3-030-00801-7_5

the language, supported by constraint solvers and non-deterministic execution on a symbolic Java virtual machine (JVM). The symbolic JVM adapts concepts from the Warren Abstract Machine, such as choice points and trail [22]. Muli's tight integration of both paradigms facilitates development of applications whose business logic is implemented in Java, but which also require occasional search, such as operations research applications [8].

In this paper, we describe a reduction semantics for a core subset of Muli. In particular, the interaction of imperative statements, free variables, and non-determinism is of interest. For simplicity, this core language abstracts from inheritance, multi-threading, and reflection, because those features do not exhibit interesting behaviour w.r.t. our semantics. The formulated semantics is helpful to get an understanding of the mechanics behind concepts that are novel to imperative and OO programming, and serves as a formal basis for implementing the symbolic JVM. It can also be used for reasoning about applications developed in Muli.

To that end, our paper is structured as follows. We provide an overview of the new language and its concepts in Sect. 2. Section 3 formalises the operational semantics of the core language. An example evaluation using this semantics is shown in Sect. 4. Section 5 presents a discussion of our concepts. Related work is outlined in Sect. 6. We then conclude in Sect. 7 and provide an outlook towards further research.

2 Language Concepts

The Muli language is derived from Java 8. We do not change existing concepts and features of Java, so that Muli also benefits from Java's well-known and well-received features, such as OO and managed memory. Instead, the language is defined by its additions to Java, i.e. Muli is a superset of Java.

Muli adds the concept of *free variables*, i.e. variables that are declared and instantiated, but not to a particular value. Instead, they are treated symbolically and can be used in statements and expressions. *Constraints* on symbolic variables and expressions are imposed during symbolic execution of conditional statements. For example, an `if` statement with a condition that involves insufficiently constrained variables results in multiple branches that can be evaluated. Conceptually, we can non-deterministically choose a branch and evaluate it. Our implementation considers all these branches using backtracking and a (complete!) iterative deepening depth-first search strategy. This is supported by a specialised symbolic JVM that records choice points for each non-deterministic branch.

Furthermore, we enforce that non-determinism only takes place inside *encapsulated search* regions, whereas code outside encapsulation is executed deterministically. This ensures that non-determinism is not introduced by accident, intending not to harm the understanding of known Java concepts. Furthermore, this ensures that the overall application exits in a single state. In contrast, unencapsulated symbolic execution could result in multiple exit states, which could

cause difficulties on the side of the caller. Encapsulation is expressed by using either of the `getAllSolutions` and `getOneSolution` operators. The logic of encapsulated search is described by *search regions* that are implemented as methods, e.g. as lambda abstractions, in order to defer their evaluation until encapsulation begins.

Solutions of encapsulated search are defined by values or expressions returned from search regions. Due to non-determinism, multiple solutions can be returned from search. Additionally, we introduce the special statement **`fail;`**, whose evaluation results in immediate backtracking in the symbolic JVM without recording a solution for the current branch.

From a syntactic perspective, these concepts extend Java only minimally. The resulting syntax of Muli can best be demonstrated using an example. Listing 1 exhibits a Muli method `log()` that searches for the logarithm of a number x to the base 2 using a free variable y and a method `pow` that calculates b^y imperatively, which is constrained to be equal to x.

```
int log(int x) {
    int y free;
    if (pow(2,y) == x) return y;
    else fail; }
int pow(int b, int y) {
    int i; int r; i = 0; r = 1;
    while (i < y) {
        r = r * b; i = i + 1; }
    return r; }
```

Listing 1. Non-deterministic computation of the logarithm of a number to the base 2 using (core) Muli.

Let us assume that the considered search region consists of a call to `log`, e.g. `log(4)`. When calling `log` with a given x, the free variable y is created and then passed to `pow` that calculates the power b^y symbolically, as y is free. Therefore, it returns a value that is accompanied by a set of accumulated constraints from which this particular value follows.[1] Consequently, `log` computes the logarithm by defining a constraint system using an imperative method that calculates the power.

If the variables involved in a branching condition (of **`if`** or **`while`** in Listing 1) are not sufficiently constrained, one of the feasible branches is chosen non-deterministically. Actually, our symbolic JVM would try them systematically one after the other, aided by a backtracking mechanism. When selecting a branch, the corresponding condition is added to the constraint store and consistency is checked. For example, **`while`** `(i < y)` can be either true or false as y is a free variable. As a result, one branch assumes the condition to be true and therefore adds the constraint $i < y$ to the constraint store by imposing a conjunction of the existing store and the new constraint. In contrast, the second branch

[1] In other problems the return value could be a symbolic expressions if the accumulated constraints do not reduce the return value's domain to a concrete value.

assumes it to be false and therefore adds the negated condition as a constraint. If an added constraint renders the store inconsistent, backtracking occurs, i.e. that branch is pruned and execution continues with a subsequent branch. Similarly, backtracking occurs when a solution is found so that the next branch can be evaluated to find further solutions. Muli's encapsulated search operators use lazy streams to return collected solutions to the surrounding deterministic computation, such that the surrounding computation can decide how many solutions it wants to obtain.

3 A Non-deterministic Operational Semantics of Muli

Muli is an extension to Java and therefore intends to fully support all Java functionality. In fact, all Muli programs even compile to regular JVM bytecode that can be parsed and executed by a regular JVM (but incorrectly), and all Java programs can be executed correctly by Muli's symbolic JVM. Outside of encapsulated search, execution in Muli is deterministic and replicates the behaviour of a standard JVM [12]. Inside encapsulation, search regions are executed non-deterministically. This changes the semantics of Java and adds subtleties that need to be explicated, particularly regarding the interaction of imperative statements, free variables, and non-determinism. Therefore, we formally define the semantics for non-deterministic evaluation of search regions.

For the purpose of describing a (non-deterministic) operational semantics of Muli, we focus on an imperative, procedural subset of Java (and Muli). This concise subset allows us to focus on the interaction between imperative and constraint-logic programming. It therefore abstracts from some features that are expected from Java but that would not contribute to the discussion in the present paper, such as inheritance.[2] Furthermore, this semantics abstracts from the execution of deterministic program parts and therefore does not prescribe an implementation for the encapsulation operators, `getAllSolutions` and `getOneSolution`.

Let us first describe the syntax of our core language. We will use variables taken from a finite set $Var = \{x_1, \ldots, x_m\}$, for simplicity all of type integer ($m \in \mathbb{N}$). Also let $Op = AOp \cup BOp \cup ROp = \{+, -, *, /\} \cup \{\&\&, ||\} \cup \{==, !=, <=, >=, <, >\}$ be a finite set of arithmetic, boolean, and relational operation symbols, respectively. We focus on binary operation symbols. Furthermore, $\mathcal{M}$ is a finite set of methods.[3]

The syntax of arithmetic expressions and boolean expressions as well as statements can be described by the following grammar. $AExpr$, $BExpr$, and $Stat$ denote the sets of all arithmetic expressions, boolean expressions, and statements, respectively, which can be constructed by the rules of this grammar.

[2] Nevertheless, Muli's symbolic JVM supports these features exactly according to the JVM specification [12] (but does not add interesting details w.r.t. non-determinism).

[3] In fact they are functions, since we ignore object-orientation in this presentation.

$e ::= c \mid x \mid e_1 \oplus e_2 \mid m(e_1, \ldots, e_k)$
where $c \in \mathbb{Z}$, $x \in Var$, $e_1, \ldots, e_k \in AExpr$, $\oplus \in AOp$, $m \in \mathcal{M}$, $k \in \mathbb{N}$,

$b ::= e_1 \odot e_2 \mid b_1 \otimes b_2 \mid \texttt{true} \mid \texttt{false}$
where $e_1, e_2 \in AExpr$, $b_1, b_2 \in BExpr$, $\odot \in ROp$, $\otimes \in BOp$,

$s ::= ; \mid \texttt{int}\ x; \mid \texttt{int}\ x\ \texttt{free}; \mid x = e; \mid e; \mid \{s\} \mid s_1\ s_2 \mid$
$\texttt{if}\ (b)\ s_1\ \texttt{else}\ s_2 \mid \texttt{while}\ (b)\ s \mid \texttt{return}\ e; \mid \texttt{fail};$
where $x \in Var$, $e \in AExpr$, $b \in BExpr$, $s, s_1, s_2 \in Stat$.

Note, in particular, the possibility to create free logic variables by `int` x `free`;.

After describing the syntax of the core language, let us now define its semantics. In the sequel, let $\mathcal{A} = \{\alpha_0, \ldots, \alpha_n\}$ be a finite set of memory addresses ($n \in \mathbb{N}$). Moreover, let

$$Tree(\mathcal{A}, \mathbb{Z}) = \mathcal{A} \cup \mathbb{Z} \cup \{\oplus(t_1, t_2) \mid t_1, t_2 \in Tree(\mathcal{A}, \mathbb{Z}), \oplus \in Op\}$$

be the set of all symbolic expression trees with addresses and integer constants as leaves and operation symbols as internal nodes.

We provide a reduction semantics, where the computations depend on an environment, a state, and a constraint store. Let $Env = (Var \cup \mathcal{M}) \to (\mathcal{A} \cup (Var^* \times Stat))$ be the set of all environments, mapping each variable to an address and each function to a representation $((x_1, \ldots, x_k), s)$ that describes its parameters and code, with the additional restriction that elements of Env may neither map variables to parameters and code nor functions to addresses. We consider functions to be in global scope and define a special initial environment $\rho_0 \in Env$ that maps functions to their respective parameters and code. Moreover, let $\Sigma = \mathcal{A} \to (\{\bot\} \cup Tree(\mathcal{A}, \mathbb{Z}))$ be the set of all possible memory states. In $\sigma \in \Sigma$, a special address α_0 with $\sigma(\alpha_0) = \bot$ is reserved for holding return values of method invocations. Furthermore, $CS = \{\texttt{true}\} \cup Tree(\mathcal{A}, \mathbb{Z})$ is the set of all possible constraint store states. Since constraints are specific boolean expressions, only conjunctions and relational operation symbols such as == and > will appear at the root of such a tree.

In the sequel, $\rho \in Env$, $\sigma \in \Sigma$, $\gamma \in CS$; if needed, we will also add discriminating indices. We will use the notation $a[x/d]$ when modifying a state or environment a, meaning

$$a[x/d](b) = \begin{cases} d & , \text{ if } b = x \\ a(b) & , \text{ otherwise.} \end{cases}$$

A free variable is represented by a reference to its own location in memory. Consequently, $\sigma(\rho(x)) = \rho(x)$ if x is a free variable. Initially, a constraint store γ is empty, i.e. it is initialised with `true`. During execution of a program, constraints may incrementally be added to the store. This is done by imposing a conjunction of the existing constraints and a new constraint, thus replacing the constraint store by the new conjunction. As a result, the constraint store is typically described by a conjunction of atomic boolean expressions. We treat the constraint solver as a black box. In our implementation, we use the external

constraint solver JaCoP [11] in its most recent version 4.4. In fact, the constraint solver is exchangeable and any solver implementation fulfilling our requirements (particularly incremental adding/removal of constraints) can be used.[4]

Note that our definition of functions does not fully cover the concept of methods in object-oriented languages, since we abstract from classes and, therefore, inheritance. However, a function in our semantics can be compared to a static method, since a function in this semantics can access and modify its own arguments and variables, but not instance variables of an object. Static fields could be modelled as global variables, i.e. further entries in ρ_0.

Since classes, inheritance, instance variables, and static variables have little influence on the interaction between imperative statements, free variables, and non-determinism, object orientation can be considered (almost) orthogonal to our work.

3.1 Semantics of Expressions

Let us start with the semantics of expressions. The semantics of expressions is described by a relation $\rightarrow \subset (Expr \times Env \times \Sigma \times CS) \times ((\mathbb{B} \cup Tree(\mathcal{A}, \mathbb{Z})) \times \Sigma \times CS)$, which we use in infix notation. Note that evaluating an expression can, in general, change state and constraint store as a side effect, although only the Invoke rule actively does so. We will point out expressions that make use of this, whereas the others merely propagate changes (if any) resulting from the evaluation of subexpressions.

The treatment of constants and variables is trivial.

$$\langle c, \rho, \sigma, \gamma \rangle \rightarrow (c, \sigma, \gamma), \text{ if } c \in \mathbb{Z} \cup \mathbb{B} \tag{Con}$$

$$\langle x, \rho, \sigma, \gamma \rangle \rightarrow (\sigma(\rho(x)), \sigma, \gamma) \tag{Var}$$

Nested arithmetic expressions without free variables are evaluated directly, whereas expressions comprising free variables result in a (deterministic) unevaluated (!) symbolic expression ($\in Tree(\mathcal{A}, \mathbb{Z})$).

$$\frac{\langle e_1, \rho, \sigma, \gamma \rangle \rightarrow (v_1, \sigma_1, \gamma_1),\ \langle e_2, \rho, \sigma_1, \gamma_1 \rangle \rightarrow (v_2, \sigma_2, \gamma_2),\ v_1, v_2, v = v_1 \oplus v_2 \in \mathbb{Z}}{\langle e_1 \oplus e_2, \rho, \sigma, \gamma \rangle \rightarrow (v, \sigma_2, \gamma_2)} \tag{AOp1}$$

$$\frac{\langle e_1, \rho, \sigma, \gamma \rangle \rightarrow (v_1, \sigma_1, \gamma_1),\ \langle e_2, \rho, \sigma_1, \gamma_1 \rangle \rightarrow (v_2, \sigma_2, \gamma_2),\ \{v_1, v_2\} \nsubseteq \mathbb{Z}}{\langle e_1 \oplus e_2, \rho, \sigma, \gamma \rangle \rightarrow (\oplus(v_1, v_2), \sigma_2, \gamma_2)} \tag{AOp2}$$

A boolean expression of the form $e_1 \odot e_2$ is evaluated analogously.

[4] A very simple constraint solver could just take equality constraints into account. In this case, $\gamma \models x == v$, if $\gamma = b_1 \wedge \ldots \wedge b_k$ and for some $j \in \{1, \ldots, k\}$ $b_k = (x == v)$.

Coherent with Java, conjunctions of boolean expressions are evaluated non-strictly. The rules for the non-strict boolean disjunction operator || are defined analogously to the following rules for `&&`.

$$\frac{\langle b_1, \rho, \sigma, \gamma\rangle \to (v_1, \sigma_1, \gamma_1), \quad \gamma \models \neg v_1}{\langle b_1 \ \texttt{\&\&}\ b_2, \rho, \sigma, \gamma\rangle \to (\texttt{false}, \sigma_1, \gamma_1)} \tag{And1}$$

$$\frac{\langle b_1, \rho, \sigma, \gamma\rangle \to (v_1, \sigma_1, \gamma_1), \quad \gamma \not\models \neg v_1, \quad (b_2, \sigma_1, \gamma_1) \to (v_2, \sigma_2, \gamma_2)}{\langle b_1 \ \texttt{\&\&}\ b_2, \rho, \sigma, \gamma\rangle \to (\wedge(v_1, v_2), \sigma_2, \gamma_2)} \tag{And2}$$

We consider a function invocation to be an expression as well, as the caller can use its result in a surrounding expression. Evaluation of the function is likely to result in a state change as well as in additions to the constraint store. Invoking m implies that its description $\rho(m)$ is looked up and corresponding fresh addresses $\alpha_1, \ldots, \alpha_k$, one for each of its k parameters, are created. The corresponding memory locations are initialised by the caller. Note that the respective values can contain free variables. $\sigma_{k+1}(\alpha_0)$ will contain the return value from evaluating the `return` statement in the body, whose semantics will be defined later (cf. rule Ret). As the compiler enforces the presence of a `return` statement, we can safely assume that $\sigma_{k+1}(\alpha_0)$ holds a value after reducing s. Invoke resets that value to $\perp$ for further evaluations within the calling method. We use the shorthand notation $\bar{a}_k = (a_1, \ldots, a_k)$ for vectors of k elements.

$$\frac{\begin{array}{c}\langle e_1, \rho, \sigma, \gamma\rangle \to (v_1, \sigma_1, \gamma_1), \quad \langle e_2, \rho, \sigma_1, \gamma_1\rangle \to (v_2, \sigma_2, \gamma_2), \ldots, \\ \langle e_k, \rho, \sigma_{k-1}, \gamma_{k-1}\rangle \to (v_k, \sigma_k, \gamma_k), \ \rho(m) = (\bar{x}_k, \ s), \\ \langle s, \rho_0[\bar{x}_k/\bar{\alpha}_k], \sigma_k[\bar{\alpha}_k/\bar{v}_k], \gamma_k\rangle \rightsquigarrow (\rho_{k+1}, \sigma_{k+1}, \gamma_{k+1}), \ \sigma_{k+1}(\alpha_0) = r\end{array}}{\langle m(e_1, \ldots, e_k), \rho, \sigma, \gamma\rangle \ \to (r, \sigma_{k+1}[\alpha_0/\perp], \gamma_{k+1})} \tag{Invoke}$$

3.2 Semantics of Statements

Next, we describe the semantics of statements by a relation $\rightsquigarrow \subset (Stat \times Env \times \Sigma \times CS) \times (Env \times \Sigma \times CS)$, which we also use in infix notation.

A variable declaration changes the environment by reserving a fresh memory location α for that variable. A free variable is represented by a reference to its own location. Enclosing declarations in a block ensures that changes of the environment stay local.

$$\langle \texttt{int}\ x;, \rho, \sigma, \gamma\rangle \rightsquigarrow (\rho[x/\alpha], \sigma, \gamma) \tag{Decl}$$

$$\langle \texttt{int}\ x\ \texttt{free};, \rho, \sigma, \gamma\rangle \rightsquigarrow (\rho[x/\alpha], \sigma[\alpha/\alpha], \gamma) \tag{Free}$$

$$\frac{\langle s, \rho, \sigma, \gamma\rangle \rightsquigarrow (\rho_1, \sigma_1, \gamma_1)}{\langle \{\ s\ \}, \rho, \sigma, \gamma\rangle \rightsquigarrow (\rho, \sigma_1, \gamma_1)} \tag{Block}$$

As a particularity of a constraint-logic OO language, an assignment `x = e` cannot just overwrite a location in memory corresponding to `x`, since this might

have an unwanted side effect on constraints that involve `x` and refer to its former value. This side effect might turn such constraints unsatisfiable after they have been imposed and checked, thus leaving a currently executed branch in an inconsistent state. We avoid this by assigning a new memory address α_1 to the variable on the left-hand side. At the new address, we store the result from evaluating the right-hand side. Consequently, old constraints or expressions that involve the former value of `x` are deliberately left untouched by the assignment. In contrast, later uses of the variable refer to its new value. The environment is updated to achieve this behaviour.

$$\frac{\langle e, \rho, \sigma, \gamma\rangle \to (v, \sigma_1, \gamma_1)}{\langle x = e, \rho, \sigma, \gamma\rangle \rightsquigarrow (\rho[x/\alpha_1], \sigma_1[\alpha_1/v], \gamma_1)} \quad \text{(Assign)}$$

Since the syntax does not enforce that no statements follow a `return` statement, we provide sequence rules that take into account that the state may hold a value in α_0 (indicating a preceding `return`) or not ($\bot$). Further statements are executed iff the latter is the case. Otherwise, further statements are discarded as a preceding `return` has already provided a result in α_0.

$$\frac{\langle s_1, \rho, \sigma, \gamma\rangle \rightsquigarrow (\rho_1, \sigma_1, \gamma_1), \quad \sigma_1(\alpha_0) == \bot, \quad \langle s_2, \rho_1, \sigma_1, \gamma_1\rangle \rightsquigarrow (\rho_2, \sigma_2, \gamma_2)}{\langle s_1 \;\; s_2, \rho, \sigma, \gamma\rangle \rightsquigarrow (\rho_2, \sigma_2, \gamma_2)} \quad \text{(Seq)}$$

$$\frac{\langle s_1, \rho, \sigma, \gamma\rangle \rightsquigarrow (\rho_1, \sigma_1, \gamma_1), \quad \sigma_1(\alpha_0) \neq \bot}{\langle s_1 \;\; s_2, \rho, \sigma, \gamma\rangle \rightsquigarrow (\rho_1, \sigma_1, \gamma_1)} \quad \text{(SeqFin)}$$

The two following rules for if-statements introduce non-determinism in case that the constraints neither entail the branching condition nor its negation.[5]

$$\frac{\langle b, \rho, \sigma, \gamma\rangle \to (v, \sigma_1, \gamma_1), \quad \gamma_1 \not\models \neg v, \quad \langle s_1, \rho, \sigma_1, \gamma_1 \wedge v\rangle \rightsquigarrow (\rho_1, \sigma_2, \gamma_2)}{\langle \texttt{if}\ (b)\ s_1\ \texttt{else}\ s_2, \rho, \sigma, \gamma\rangle \rightsquigarrow (\rho_1, \sigma_2, \gamma_2)} \quad (\text{If}_t)$$

$$\frac{\langle b, \rho, \sigma, \gamma\rangle \to (v, \sigma_1, \gamma_1), \quad \gamma_1 \not\models v, \quad \langle s_2, \rho, \sigma_1, \gamma_1 \wedge \neg v\rangle \rightsquigarrow (\rho_1, \sigma_2, \gamma_2)}{\langle \texttt{if}\ (b)\ s_1\ \texttt{else}\ s_2, \rho, \sigma, \gamma\rangle \rightsquigarrow (\rho_1, \sigma_2, \gamma_2)} \quad (\text{If}_f)$$

As with `if`, `while` can also behave non-deterministically.

$$\frac{\langle b, \rho, \sigma, \gamma\rangle \to (v, \sigma_1, \gamma_1), \quad \gamma_1 \not\models \neg v, \quad \langle s, \rho, \sigma_1, \gamma_1 \wedge v\rangle \rightsquigarrow (\rho_1, \sigma_2, \gamma_2), \quad \langle \texttt{while}\ (b)\ s, \rho_1, \sigma_2, \gamma_2\rangle \rightsquigarrow (\rho_2, \sigma_3, \gamma_3)}{\langle \texttt{while}\ (b)\ s, \rho, \sigma, \gamma\rangle \rightsquigarrow (\rho_2, \sigma_3, \gamma_3)} \quad (\text{Wh}_t)$$

$$\frac{\langle b, \rho, \sigma, \gamma\rangle \to (v, \sigma_1, \gamma_1), \quad \gamma_1 \not\models v}{\langle \texttt{while}\ (b)\ s, \rho, \sigma, \gamma\rangle \rightsquigarrow (\rho, \sigma_1, \gamma_1 \wedge \neg v)} \quad (\text{Wh}_f)$$

[5] In the implementation, the applicability of these rules will depend on the constraint propagation abilities of the employed constraint solver. We discuss the implications in Sect. 5.

All branching rules If_f, If_t, Wh_f, and Wh_t could be accompanied by more efficient ones that deterministically choose a branch if its condition does not involve free variables, i.e. without having to consult the constraint store. We omit these rules in an effort to keep our definitions concise, as the provided ones can also handle these cases.

We assume that the code of a user-defined function is terminated by a `return` statement, i.e. its existence has to be ensured by the compiler. The corresponding return value is supplied to the caller by storing it in α_0, causing remaining statements of the function to be skipped (cf. rule SeqFin), and letting the caller extract the result from α_0 (cf. rule Invoke). The `return` statement is handled as follows:

$$\frac{\langle e, \rho, \sigma, \gamma\rangle \to (v, \sigma_1, \gamma_1)}{\langle \texttt{return}\ e, \rho, \sigma, \gamma\rangle \rightsquigarrow (\rho, \sigma_1[\alpha_0/v], \gamma_1)} \tag{Ret}$$

Furthermore, we do not define an evaluation rule involving a `fail` statement. This is intentional, as the evaluation of such a statement leads to a computation that fails immediately.

The following (optional) substitution rule allows to simplify expressions and results.

$$\frac{\gamma \models \gamma(\alpha) == v, \quad \langle s, \rho, \sigma[\alpha/v], \gamma\rangle \rightsquigarrow (\rho_1, \sigma_1, \gamma_1)}{\langle s, \rho, \sigma, \gamma\rangle \rightsquigarrow (\rho_1, \sigma_1, \gamma_1)} \tag{Subst}$$

When variables are not sufficiently constrained to concrete values, *labeling* can be used to substitute variables for values that satisfy the imposed constraints [5]. This non-deterministic rule is applied with the least priority, i.e. it should only be used if no other rule can be applied. Otherwise, it would result in a lot of non-deterministic branching, thus preventing the constraint solver from an efficient reduction of the search space by constraint propagation.

$$\frac{\gamma \not\models \sigma(\alpha) \neq v, \quad \langle s, \rho, \sigma[\alpha/v], \gamma \wedge (\sigma(\alpha) == v)\rangle \rightsquigarrow (\rho_1, \sigma_1, \gamma_1)}{\langle s, \rho, \sigma, \gamma\rangle \rightsquigarrow (\rho_1, \sigma_1, \gamma_1)} \tag{Label}$$

4 Example Evaluation

We demonstrate the use of the reduction rules defined in Sect. 3 by computing one possible result of the logarithm program from Listing 1 that will be invoked by an additional method `int main() { return log(1); }`. Other possible results can be computed analogously. We abbreviate the code of `log` and `pow` by s_1 and s_3, respectively, to improve readability. The substatement s_2 is included in s_1, while s_3 includes the substatements s_4, s_5, and s_6. Moreover, we use the infix notation for nested expressions, e.g. we write $n \geq 1$ instead of $\geq (n, 1)$.

Initially, let $\rho_0 = \{main \mapsto (\epsilon, \texttt{return log(1);}),\ log \mapsto ((x), s_1),\ pow \mapsto ((b, y), s_3)\}$. Furthermore, let $\gamma_1 = true$ and $\sigma_0 = \{\alpha_0 \mapsto \bot\}$. We begin in method `main()`, which evaluates to

$$\dfrac{\dfrac{\langle 1, \rho_0, \sigma_0, \gamma_1\rangle \to (1, \sigma_0, \gamma_1)\ \text{(Con)},\ \rho_0(log) = ((x), s_1),\ \sigma_6(\alpha_0) = 0}{\langle \texttt{log(1)}, \rho_0, \sigma_0, \gamma_1\rangle \to (0, \sigma_6[\alpha_0/\bot], \alpha_2 == 0)}\ \text{(Lemma}_1\text{)}\ \text{(Invoke)}}{\langle \texttt{return log(1)}, \rho_0, \sigma_0, \gamma_1\rangle \rightsquigarrow (\rho_0, \sigma_6[\alpha_0/0], \alpha_2 == 0)}\ \text{(Ret)}$$

Performing an entire evaluation with this example is interesting, but lengthy. We therefore moved the detailed evaluation into the appendix (cf. Lemma$_1$) and use the opportunity to highlight some interesting evaluation steps here. In the final state, $\sigma_6 = \sigma_0[\alpha_0/0, \alpha_1/1, \alpha_2/\alpha_2, \alpha_3/2, \alpha_4/\alpha_2, \alpha_7/0, \alpha_8/1]$.

The final result $\sigma_6(\alpha_0) = 0$ results from the constraint $\alpha_2 \leq 0$ obtained from evaluating Wh_f (Lemma$_9$ in the appendix provides context):

$$\dfrac{\dfrac{\langle \texttt{i}, \rho_4, \sigma_3, \gamma_1\rangle \to (0, \sigma_3, \gamma_1)\ \text{(Var)},\ \langle \texttt{y}, \rho_4, \sigma_3, \gamma_1\rangle \to (\alpha_2, \sigma_3, \gamma_1)\ \text{(Var)}}{\langle \texttt{i < y}, \rho_4, \sigma_3, \gamma_1\rangle \to (0 < \alpha_2, \sigma_3, \gamma_1)}\ \text{(AOp2)},\ \gamma \not\models (0 < \alpha_2)}{\langle \texttt{while (i < y) s}_6, \rho_4, \sigma_3, \gamma_1\rangle \rightsquigarrow (\rho_4, \sigma_3, \gamma_1 \wedge \neg(0 < \alpha_2))}\ (\text{Wh}_f)$$

where $\rho_4 = \rho_0[b/\alpha_3, y/\alpha_4, i/\alpha_7, r/\alpha_8]$ and $\sigma_0[\alpha_1/1, \alpha_2/\alpha_2, \alpha_3/2, \alpha_4/\alpha_2, \alpha_7/0, \alpha_8/1]$. $\alpha_2 \leq 0$ is further refined to $\alpha_2 == 0$ by the labeling rule in Lemma$_2$ in the appendix.

In Lemma$_2$, the constraint store is used to deduce that $\alpha_2 == 0$ is consistent with the current constraint, $\alpha_2 \leq 0$, as well as with the constraint store γ_2. Therefore, labeling non-deterministically imposes the more restrictive constraint $\alpha_2 == 0$. Other branches may impose further constraints consistent with $\alpha_2 < 0$.

If we had non-deterministically chosen rule Wh_t in Lemma$_9$, we would have performed an iteration of the `while` loop, leading to more computations that would not result in solutions, as they would be discarded as incorrect by the `fail` statement of the `log` method.

The evaluation of rule Assign in Lemma$_7$ creates a new memory location α_7 in σ_2 for the new value of `i` and updates the environment accordingly. At this point, no references to the old location α_5 exist, so an implementation could use garbage collection to free that location. Hypothetically, if rule Wh_t had been chosen in Lemma$_9$, an iteration of the loop would have resulted in additional evaluations of rule Assign, e.g. to increment `i`, thus reserving additional locations. In the case of `i`, the new value would depend on the value in α_7. However, as the old value and the increment are constant, the new value would be computed by evaluating rule AOp1, so that, again, no reference to α_7 is needed.

5 Discussion

The key aspect of the semantic rules for the presented core language is the interaction between constraint-logic programming and imperative programming. Some aspects of it offer themselves for thorough discussion.

The (potentially) non-deterministic evaluation of our rules If_f, If_t, Wh_f, and Wh_t highly depends on the included constraint solver. Our definition allows to follow a branch if the negation of its condition is not entailed by the current constraint store γ. When implementing this, a constraint solver will be used to check whether $\gamma \not\models \neg v$ (analogously for $\gamma \not\models v$). If the constraint solver is not able to show that the constraints entail $\neg v$, this may have three reasons: (1) $\gamma \models v$, or (2) the current constraints neither entail v nor $\neg v$, or (3) the constraint propagation abilities of the employed constraint solver are insufficient to show that $\gamma \models \neg v$, but in fact $\gamma \models \neg v$. In case (1), the system behaves deterministically and only one rule for `if` (or `while`) will be applied. In case (2), one of the two rules for `if` (or `while`) can be chosen non-deterministically. Only case (3) is problematic. In this case, a branch can be chosen that corresponds to inconsistent constraints. In practice, solvers do not achieve perfect constraint propagation and also no global consistency of the constraints. Consequently, results corresponding to inconsistent constraints may only be discovered later, e.g. during labeling. In the meantime, non-backtrackable statements (e.g. ones that result in input/output) of search regions may have been executed in branches that prove infeasible later. Thus, we suggest to avoid input/output in search regions.

We would like to point out that the aforementioned problem is not specific to Muli, as this can occur in Prolog (using CLP(FD) [20]) as well. Consider the Prolog program provided in Listing 2. When you execute the first goal, the output will (among the unreduced constraint system) contain a line that says `successful`, even though it is apparent to the human reader that there is no solution, so that the `write` statement should not have been reached. In contrast, if `label` is invoked before `write` (second goal), Prolog realises that there is no solution and therefore gives the correct result `false`.

```
use_module(library(clpfd)).
?- [X,Y,Z] ins 0..1, all_different([X,Y,Z]),
     write('successful').
?- [X,Y,Z] ins 0..1, all_different([X,Y,Z]),
     label([X,Y,Z]), write('successful').
```

Listing 2. Demonstration of the limits of constraint propagation using an example in Prolog+CLP(FD).

We see two options to handle this situation in Muli programs. The first option is to explicitly label variables sufficiently at every branch such that the constraint solver is able to either infer $\gamma \models v$ or $\gamma \models \neg v$. However, as explained in context of the Label rule, this also introduces a lot of non-deterministic branching by creating one branch per label. Therefore, the effectivity of constraint propagation is reduced and the overall effort for search is increased. For the same reason we

decided that Muli should not implicitly perform labeling at every branch either, as performance would deteriorate.

The second option is to perform labeling only after a solution has been found during encapsulated search. In fact, such a solution is merely a potential solution, under the condition that the corresponding constraints are also satisfiable. As a result, encapsulated search produces a stream of pairs, each of which comprises one potential solution and its corresponding set of constraints. Thus, at this point the enclosing application can iterate over this stream and perform (sufficient) labeling, until it is clear whether the constraints are actually satisfiable. This rules out infeasible solutions afterwards. The implementation of Muli provides an explicit `label` operation, which the application developer can use for this purpose. We decided not to do this implicitly in order to give the developer more flexibility. It is easy to wrap this functionality into a search operation which labels every found solution implicitly.

Both mentioned options are available to the developer. We recommend the second one, possibly in the wrapped version with implicit labeling. For search regions that involve only backtrackable statements, the result does not depend on the chosen option, but the second option is presumably more efficient as fewer branches have to be evaluated. For other search regions, only the first option can avoid unwanted side effects of illegally accessed branches. However, search then becomes less efficient. Therefore, in case that non-backtrackable side effects have to be avoided, we recommend that the developer removes input/output operations from search regions and moves them behind encapsulation instead.

Formalising the operational semantics of Muli has also helped uncover some operations whose semantics are sufficiently clear in deterministic Java, but become ambiguous when non-determinism and symbolic execution are added. Consequently, some alternatives could be discussed on a conceptual level using this semantics, before deriving a corresponding implementation. This particularly involves the interpretation of symbolic variables (rules Invoke and Var) and assignments, as outlined subsequently.

By rule Assign, an assignment `x = e` creates a new memory address for the variable `x` and changes the environment accordingly. As a result, memory usage of a Muli program is increased with every assignment, instead of with every declaration of a variable as in imperative OO languages. Nevertheless, this behaviour is required in order to avoid unwanted side effects on previous constraints involving `x`. The alternative, mutating $\sigma(\rho(x))$ directly, would result in assignments to `x` that could render constraints involving `x` unsatisfiable ex post, i.e. after branching has occurred that depended on such a constraint.

As another consequence, rule Assign ensures that the interpretation of symbolic variables is equivalent to that of regular values. Consider the simple excerpt from a Java program given in Listing 3 as an example: After evaluating the last line, `y` is still expected to be 5, even though `x` now holds a different value. After all, although primitive variables can be directly mutated in Java, their previous interpretations cannot. Similarly, for symbolic values, rule Assign ensures that references before and after an assignment are treated distinctly, even though

memory efficiency is adversely affected. Nevertheless, unreferenced former meanings of a variable may be destroyed by the garbage collector, thus reclaiming (some) memory.

```
int x = 5; int y = x;
x = 3;
```

Listing 3. Minimal example demonstrating that variables may be mutated directly, in contrast to results of their uses: After evaluation, `y` is 5.

Implicitly, our rules Assign (or Invoke) and Var enable sharing of symbolic values. Assigning a free variable x to another free variable y means that the address $\rho(x)$ of x is stored in the memory location corresponding to y by modifying state as $\sigma[\rho(y)/\rho(x)]$. Consequently, subsequent constraints and expressions that involve either variable will actually reference the same variable. The sharing behaviour is exhibited in the example in Lemma_5 in the appendix, where a free variable is passed to the `pow` method as its second parameter. `pow` adds constraints to that variable that only come into effect when labeling is performed in its invoking context in `log` (Lemma_2 in the appendix).

Regarding backtracking, the implementation is only implicitly affected by the presented operational semantics. Here, the semantics defines the desired state of the overall VM that must be achieved before evaluation in terms of $\rho \in Env$, $\sigma \in \Sigma$, $\gamma \in CS$. Considering the multitude of options for achieving the desired VM state that lend themselves for the implementation, we briefly outline the options without prescribing either. Firstly, "don't care" non-determinism considers only one evaluation alternative and therefore does not require backtracking at all. Secondly, it would be possible to fork at statements that introduce non-determinism, thus evaluating all alternatives in parallel. This does not require backtracking either, however, consider that this generates a lot of overhead in terms of memory and computation, as the VM must be forked in its entirety to accommodate for any side effects, and as all forks must be joined in order to return to deterministic computation after a search region is fully processed. Thirdly, the alternatives can be evaluated sequentially. To achieve this, the VM must record changes to the data structures on a trail equivalent to that of Prolog in order to reconstruct a previous state during backtracking. Our implementation resorts to the latter option using a trail adapted from the Warren Abstract Machine. Nevertheless, the remaining options would also be interesting to pursue.

6 Related Work

To the best of our knowledge, this paper is the first to present a formal semantics of an imperative language enhanced by features of constraint-logic programming. For sake of clarity we focused on a core language. A full formal semantics of Java alone may require an entire book as in the work by Stärk et al. [19]. K-Java [2] is another approach to define a formal semantics of Java. However, in the

cited paper the authors focus on selected aspects of the language. The official semantics of Java is extensively described in natural language (cf. [6,12]).

Some existing core languages of Java such as Featherweight Java [9] are tailored to the investigation of the typing system and not meant to be executable. Hainry [7] investigates an object-oriented core language focussing on computational complexity. As a result of their respective foci they were not suitable to be extended for Muli.

The encapsulated search of Muli has been inspired and adapted from the corresponding feature of the functional-logic language Curry. An operational semantics of Curry can be found in [1]. It is simpler than our semantics, since Curry is purely declarative and does not have to bother with side effects.

Approaches for integrating object-oriented features into a (constrained) logic language are e.g. Oz [21], Visual Prolog [17], Prolog++ [15], and Concurrent Prolog [18]. However, these approaches maintain a declarative flavour and mainly provide syntactic sugar for object-orientation. They are unfamiliar for mainstream object-oriented programmers.

There are also approaches which add constrained-logic features to an imperative/object-oriented language. Typically, the integration is less seamless than in Muli and the language parts stemming from different paradigms can clearly be distinguished [3,4]. CAPJa combines Java and Prolog and provides a simplified interface mapping Java objects to Prolog terms, but requires distinct code in each language nevertheless [16]. LogicJava [14] is more restrictive than Muli and only allows class fields to be logic variables. Moreover, entire methods have to be declared as searching or non-searching.

7 Conclusions and Future Work

Our work formalises an operational reduction semantics for an imperative core of the novel integrated constraint-logic object-oriented language Muli. Muli extends Java by logic variables, non-determinism, encapsulated search, and constraint solving. Muli is particularly suited for enterprise applications that involve both searching and non-searching business logic. Encapsulated search ensures that non-determinism is only introduced deliberately where needed, instead of spreading out over the whole program. Thus, the code outside of encapsulated search regions behaves just as ordinary Java code.

The presented operational semantics provides a basis for implementations of compiler, symbolic JVM, and tools for processing Muli programs. In particular, the formalisation has helped clarify possible ambiguities w.r.t. the semantics of certain statements under non-determinism, such as that of assignments to variables and uses of them. Furthermore, the semantics will facilitate reasoning about programs developed in Muli as demonstrated in the example evaluation. We made the symbolic JVM that executes Muli programs available as free software on GitHub.[6]

[6] https://github.com/wwu-pi/muli-env.

As future work, we would like to extend our core language and its semantics by more features of Java, such as classes and inheritance. We expect these additions to be quite orthogonal to the presently supported concepts. However, when (non-deterministically) instantiating a free variable with an object type, we have to take the whole corresponding inheritance hierarchy into account.

Appendix: Full Example Evaluation

In addition to ρ_0, σ_0, and γ_1 defined in section Sect. 4, the following auxiliary definitions will be needed as intermediate results: $\rho_1 = \rho_0[x/\alpha_1, y/\alpha_2]$, $\rho_2 = \rho_0[b/\alpha_3, y/\alpha_4]$, $\rho_3 = \rho_2[i/\alpha_5, r/\alpha_6]$, $\rho_4 = \rho_3[i/\alpha_7, r/\alpha_8]$, $\sigma_1 = \sigma_0[\alpha_1/1, \alpha_2/\alpha_2]$, $\sigma_2 = \sigma_1[\alpha_3/2, \alpha_4/\alpha_2]$, $\sigma_3 = \sigma_2[\alpha_7/0, \alpha_8/1]$, $\sigma_4 = \sigma_3[\alpha_0/1]$, $\sigma_5 = \sigma_4[\alpha_0/\ \bot]$, $\sigma_6 = \sigma_5[\alpha_0/0]$, $\gamma_2 = \gamma_1 \wedge \alpha_2 \leq 0$, and $\gamma_3 = \gamma_2 \wedge \alpha_2 == 0$. To simplify the understanding of the full computation provided in Sect. 4, we have decomposed it into a couple of lemmas. We present the computation in a top-down fashion. If you prefer a bottom-up fashion, just read the lemmas in reverse order. The names of the applied rules are specified in each step.

$$\frac{\langle \texttt{int y free;}\rangle \rightsquigarrow (\rho_0[x/\alpha_1, y/\alpha_2], \sigma_0[\alpha_1/1, \alpha_2/\alpha_2], \gamma_1)\ \text{(Free)}, \quad \text{(Lemma}_2\text{)}}{\langle \texttt{int y free; s}_2, \rho_0[x/\alpha_1], \sigma_0[\alpha_1/1], \gamma_1\rangle \rightsquigarrow (\rho_1, \sigma_6, \gamma_3)}\ \text{(Seq)} \qquad \text{(Lemma}_1\text{)}$$

$$\frac{\text{(Lemma}_3\text{)},\ \gamma_2 \not\models \neg\texttt{true},\ \dfrac{\gamma_2 \not\models \sigma_5(\alpha_2) \neq 0,\ \text{(Lemma}_4\text{)}}{\langle \texttt{return y;}, \rho_1, \sigma_5, \gamma_2\rangle \rightsquigarrow (\rho_1, \sigma_6, \gamma_3)}\ \text{(Label)}}{\langle s_2, \rho_1, \sigma_1, \gamma_1\rangle \rightsquigarrow (\rho_1, \sigma_6, \gamma_3)}\ (\text{If}_{\text{t}}) \qquad \text{(Lemma}_2\text{)}$$

$$\frac{\text{(Lemma}_5\text{)},\ \langle x, \rho_1, \sigma_5, \gamma_2\rangle \rightarrow (1, \sigma_5, \gamma_2)\ \text{(Var)},\ 1 == 1 = \texttt{true}}{\langle \texttt{pow(2, y)} == \texttt{x}, \rho_1, \sigma_1, \gamma_1\rangle \rightarrow (\texttt{true}, \sigma_5, \gamma_2)}\ \text{(AOp1)} \qquad \text{(Lemma}_3\text{)}$$

$$\frac{\langle y, \rho_1, \sigma_5[\alpha_2/0], \gamma_2 \wedge \alpha_2 == 0\rangle \rightarrow (0, \sigma_5, \gamma_3)\ \text{(Var)}}{\langle \texttt{return y;}, \rho_1, \sigma_5, \gamma_2\rangle \rightsquigarrow (\rho_1, \sigma_5[\alpha_0/0], \gamma_3)}\ \text{(Ret)} \qquad \text{(Lemma}_4\text{)}$$

$$\frac{\begin{array}{l}\langle 2, \rho_1, \sigma_1, \gamma_1\rangle \rightarrow (2, \sigma_1, \gamma_1)\ \text{(Con)},\\ \quad \langle y, \rho_1, \sigma_1, \gamma_1\rangle \rightarrow (\alpha_2, \sigma_1, \gamma_1)\ \text{(Var)},\\ \qquad \rho_1(pow) = ((b, y), s_3),\ \text{(Lemma}_6\text{)},\ \sigma_4(\alpha_0) = 1\end{array}}{\langle \texttt{pow(2, y)}, \rho_1, \sigma_1, \gamma_1\rangle \rightarrow (1, \sigma_4[\alpha_0/\bot], \gamma_2)}\ \text{(Invoke)} \qquad \text{(Lemma}_5\text{)}$$

$$\dfrac{\dfrac{\langle \texttt{int i;}, \rho_2, \sigma_2, \gamma_1\rangle \rightsquigarrow (\rho_2[i/\alpha_5], \sigma_2, \gamma_1)\ \text{(Decl)},\quad \langle \texttt{int r;}, \rho_2[i/\alpha_5], \sigma_2, \gamma_1\rangle \rightsquigarrow (\rho_2[i/\alpha_5, r/\alpha_6], \sigma_2, \gamma_1)\ \text{(Decl)},\quad \text{(Lemma}_7\text{)}}{\langle \texttt{int r; i = 0; r = 1; s}_4, \rho_2[i/\alpha_5], \sigma_2, \gamma_1\rangle \rightsquigarrow (\rho_4, \sigma_4, \gamma_2)}\ \text{(Seq)}}{\langle \texttt{int i; int r; i = 0; r = 1; s}_4, \rho_2, \sigma_2, \gamma_1\rangle \rightsquigarrow (\rho_4, \sigma_4, \gamma_2)}\ \text{(Seq)}$$

$$\text{(Lemma}_6\text{)}$$

$$\dfrac{\dfrac{\dfrac{\langle 0, \rho_3, \sigma_2, \gamma_1\rangle \rightarrow (0, \sigma_2, \gamma_1)\ \text{(Con)}}{\langle \texttt{i = 0;}, \rho_3, \sigma_2, \gamma_1\rangle \rightsquigarrow (\rho_3[i/\alpha_7], \sigma_2[\alpha_7/0], \gamma_1)}\ \text{(Assign)},\quad \dfrac{\langle 1, \rho_3[i/\alpha_7], \sigma_2[\alpha_7/0], \gamma_1\rangle \rightarrow (0, \sigma_2[\alpha_7/0], \gamma_1)\ \text{(Con)}}{\langle \texttt{r = 1;}, \rho_3[i/\alpha_7], \sigma_2[\alpha_7/0], \gamma_1\rangle \rightsquigarrow (\rho_4, \sigma_3, \gamma_1)}\ \text{(Assign)},\quad \text{(Lemma}_8\text{)}}{\langle \texttt{r = 1; s}_4, \rho_3[i/\alpha_7], \sigma_2[\alpha_7/0], \gamma_1\rangle \rightsquigarrow (\rho_4, \sigma_4, \gamma_2)}\ \text{(Seq)}}{\langle \texttt{i = 0; r = 1; s}_4, \rho_3, \sigma_2, \gamma_1\rangle \rightsquigarrow (\rho_4, \sigma_4, \gamma_2)}\ \text{(Seq)}$$

$$\text{(Lemma}_7\text{)}$$

$$\dfrac{\text{(Lemma}_9\text{)},\quad \dfrac{\langle r, \rho_4, \sigma_3, \gamma_2\rangle \rightarrow (1, \sigma_3, \gamma_2)\ \text{(Var)}}{\langle \texttt{return r;}, \rho_4, \sigma_3, \gamma_2\rangle \rightsquigarrow (\rho_4, \sigma_3[\alpha_0/1], \gamma_2)}\ \text{(Ret)}}{\langle \texttt{s}_5\texttt{; return r;}, \rho_4, \sigma_3, \gamma_1\rangle \rightsquigarrow (\rho_4, \sigma_3[\alpha_0/1], \gamma_2)}\ \text{(Seq)} \qquad \text{(Lemma}_8\text{)}$$

$$\dfrac{\dfrac{\langle \texttt{i}, \rho_4, \sigma_3, \gamma_1\rangle \rightarrow (0, \sigma_3, \gamma_1)\ \text{(Var)},\quad \dfrac{\langle \texttt{y}, \rho_4, \sigma_3, \gamma_1\rangle \rightarrow (\alpha_2, \sigma_3, \gamma_1)\ \text{(Var)}}{\langle \texttt{i < y}, \rho_4, \sigma_3, \gamma_1\rangle \rightarrow (0 < \alpha_2, \sigma_3, \gamma_1)}\ \text{(AOp2)},\quad \gamma \not\models (0 < \alpha_2)}{\langle \texttt{while (i < y) s}_6, \rho_4, \sigma_3, \gamma_1\rangle \rightsquigarrow (\rho_4, \sigma_3, \gamma_1 \wedge \neg(0 < \alpha_2))}\ \text{(Wh}_f\text{)} \qquad \text{(Lemma}_9\text{)}$$

References

1. Albert, E., Hanus, M., Huch, F., Oliver, J., Vidal, G.: An operational semantics for declarative multi-paradigm languages. Electron. Notes Theor. Comput. Sci. **70**(6), 65–86 (2002)
2. Bogdanas, D., Rosu, G.: K-Java: a complete semantics of Java. In: POPL 2015, pp. 1–12 (2015)
3. Cimadamore, M., Viroli, M.: A Prolog-oriented extension of Java programming based on generics and annotations. In: Amaral, V., et al. (eds.) Proceedings PPPJ, ACM ICPS, vol. 272, pp. 197–202. ACM (2007)
4. Cimadamore, M., Viroli, M.: Integrating Java and Prolog through generic methods and type inference. In: Wainwright, R.L., Haddad, H. (eds.) Proceedings of the 2008 SAC, pp. 198–205. ACM (2008). https://doi.org/10.1145/1363686
5. Frühwirth, T., Abdennadher, S.: Essentials of Constraint Programming. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-662-05138-2

6. Gosling, J., Joy, B., Steele, G., Bracha, G., Buckley, A.: The Java® Language Specification - Java SE 8 Edition (2015). https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf
7. Hainry, E., Péchoux, R.: Objects in polynomial time. In: Feng, X., Park, S. (eds.) APLAS 2015. LNCS, vol. 9458, pp. 387–404. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-26529-2_21
8. Hooker, J.N.: Operations research methods in constraint programming (Chap. 15). In: Rossi, F., van Beek, P., Walsh, T. (eds.) Handbook of CP. Elsevier, Amsterdam (2006)
9. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight Java: a minimal core calculus for Java and GJ. ACM Trans. Program. Lang. Syst. **23**(3), 396–450 (2001)
10. Kondoh, G., Onodera, T.: Finding bugs in Java native interface programs. In: ISSTA 2008, p. 109 (2008)
11. Kuchcinski, K.: Constraints-driven scheduling and resource assignment. ACM Trans. Des. Autom. Electron. Syst. **8**(3), 355–383 (2003)
12. Lindholm, T., Yellin, F., Bracha, G., Buckley, A.: The Java® Virtual Machine Specification - Java SE 8 Edition (2015). https://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf
13. Louden, K.C.: Programming Languages: Principles and Practice. Wadsworth Publ. Co., Belmont (1993)
14. Majchrzak, T.A., Kuchen, H.: Logic Java: combining object-oriented and logic programming. In: Kuchen, H. (ed.) WFLP 2011. LNCS, vol. 6816, pp. 122–137. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22531-4_8
15. Moss, C.: Prolog++ - The Power of Object-Oriented and Logic Programming. International Series in Logic Programming. Addison-Wesley, Boston (1994)
16. Ostermayer, L.: Seamless cooperation of Java and Prolog for rule-based software development. In: Proceedings of the RuleML 2015, Berlin (2015)
17. Scott, R.: A Guide to Artificial Intelligence with Visual Prolog. Outskirts Press (2010)
18. Shapiro, E., Takeuchi, A.: Object oriented programming in concurrent Prolog. New Gener. Comput. **1**(1), 25–48 (1983)
19. Stärk, R., Schmid, J., Börger, E.: Java and the Java Virtual Machine - Definition, Verification, Validation. Springer, Heidelberg (2001). https://doi.org/10.1007/978-3-642-59495-3
20. Triska, M.: The finite domain constraint solver of SWI-Prolog. In: Schrijvers, T., Thiemann, P. (eds.) FLOPS 2012. LNCS, vol. 7294, pp. 307–316. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-29822-6_24
21. Van Roy, P., Brand, P., Duchier, D., Haridi, S., Schulte, C., Henz, M.: Logic programming in the context of multiparadigm programming: the Oz experience. Theory Pract. Log. Program. **3**(6), 717–763 (2003)
22. Warren, D.H.D.: An abstract Prolog instruction set. Technical report, SRI International, Menlo Park (1983)