ERCIS – European Research Center for Information Systems
Westfälische Wilhelms-Universität Münster
Leonardo-Campus 3, 48149 Münster, Germany
P: +49 (0)251 83-38100 F: +49 (0)251 83-38109
E: info@ercis.org W: http://www.ercis.org/

ISSN 1614-7448

ERCIS WORKING PAPERS

Working Paper No. 32

Free Objects in Constraint-logic Object-oriented Programming

Dageförde, Jan C.
Kuchen, Herbert

Working Papers

ERCIS — European Research Center for Information Systems
Editors: J. Becker, M. Dugas, B. Hellingrath, T. Hoeren,
S. Klein, H. Kuchen, H. Trautmann, G. Vossen

Working Paper No. 32

Free Objects in Constraint-logic Object-oriented Programming

Jan C. Dageförde, Herbert Kuchen

ISSN 1614-7448

cite as: Jan C. Dageförde, Herbert Kuchen: Free Objects in Constraint-logic Object-oriented Programming. In: Working Papers, European Research Center for Information Systems No. 32. Eds.: Becker, J. et al. Münster 2020.

Contents

List of Figures

Working Paper Sketch

Type

Research Report

Title

Free Objects in Constraint-logic Object-oriented Programming.

Authors

Jan C. Dageförde, Herbert Kuchen
jan.dagefoerde@ercis.uni-muenster.de, kuchen@uni-muenster.de

Abstract

Constraint-logic object-oriented programming is useful in the integrated development of business software that occasionally solves constraint-logic problems. So far, work in constraint-logic object-oriented programming was limited to considering constraints that only involve logic variables of primitive types; in particular, boolean, integer, and floating-point numbers. However, the availability of object-oriented features calls for the option to use logic variables in lieu of objects as well. Therefore, support for reference-type logic variables (or *free objects*) is required. With the present work, we add support for reference-type logic variables to a Java-based constraint-logic object-oriented language. Allowing free objects in statements and expressions results in novel interactions with objects at runtime, for instance, non-deterministic execution of invocations on free objects (taking arbitrary class hierarchies and overriding into account). In order to achieve this, we also propose a dynamic type constraint that restricts the types of free objects at runtime.

Keywords

Constraint-logic object-oriented programming, reference-type logic variables, programming language implementation, runtime systems.

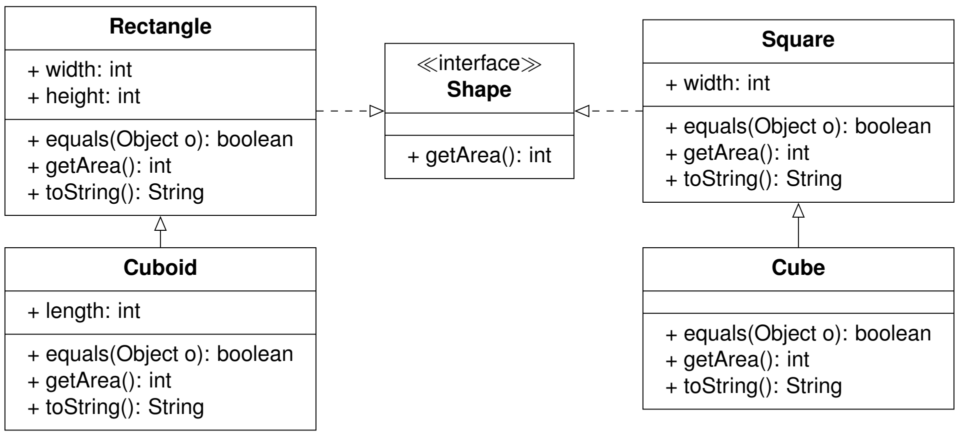


Figure 1: Class structure assumed for the running example.

1 Programming with Free Objects

With *constraint-logic object-oriented programming (CLOOP)*, software engineers are offered an integrated programming paradigm for the development of business software that occasionally requires search for solutions to constraint-logic problems. As a mixed paradigm, CLOOP provides well-known benefits of object-oriented programming languages (e. g., encapsulation of data and behaviour) as well as of constraint-logic programming (declarative specification and solving of search problems). For example, the CLOOP language *Muli* is based on Java and adds logic variables, symbolic execution, constraints, and encapsulated search. Muli code is executed by the Muli Logic Virtual Machine (MLVM), which is a custom Java virtual machine that also provides support for symbolic execution and constraint solving [DK19]. Syntactically, logic variables in Muli can be of arbitrary types. However, so far, constraints can only be defined over logic variables of *primitive* types. Primitive logic variables may still be assigned to fields of objects, so they can already be used in an object-oriented context. Nevertheless, adding support for logic variables that represent entire objects requires additional work.

Adding support for reference-type logic variables (*free objects* in particular) raises interesting questions. After all, objects in object-oriented languages (and, therefore, also in CLOOP) encapsulate data *and behaviour*. For instance, consider the following code from Listing 1 in the context of the class hierarchy illustrated in Figure 1, which will serve as a running example.

```
Shape s free;
if (s.getArea() == 16) {
  System.out.println(s.toString()); }
```

Listing 1: Excerpt from a constraint-logic object-oriented program that invokes a method on a free object.

As `Shape` merely provides the interface, the invocation of `s.getArea()` can be interpreted in multiple ways depending on the number of implementations of `Shape`. Like in this example, we generally assume that the type of a free object is only partially known, i. e., when a variable that is declared as `Object o` is of type `Object`, `o` may in fact hold an instance of `Object` or of any subtype. Consequently, there is only partial information about the (actual) type of an object, so that there are implications for

- how free objects are instantiated,

- how accesses to fields of a free object are handled,
- invocations of methods on free objects,
- type operations on free objects, and
- the notion of equality of free (and regular) objects.

Some of these implications have been discussed conceptually in a research-in-progress paper, but the discussion was incomplete and did not yet result in an implementation [Dag19].

In order to effectively realize the benefits of an integrated CLOOP language, these implications need to be discussed in order to define the semantics and to implement a runtime environment. In CLOOP, we expect the applicable alternatives to be evaluated non-deterministically until all alternatives are considered ("don't know" non-determinism) [DK19]. Therefore, the example in Listing 1 would result in at least four lines of output as there are four classes that implement the `Shape` interface.

This paper provides the following contributions to CLOOP, all of which we have exemplarily implemented in a modified MLVM:

- A semantics for non-deterministic method invocations on free objects, i. e., on objects whose type is only partially known (Section 3). This is achieved in combination with a dynamic type constraint that restricts the valid types of an object at runtime.
- A discussion of how fields of free objects are accessed (Section 4).
- Another application of the dynamic type constraint for the implementation of type operations, namely casts and checks in Section 5.1, followed by a discussion of equality of (free) objects (Section 5.2).
- Throughout the paper we use several examples that demonstrate how free objects are useful in programming. In addition, we display larger example applications for demonstration purposes in Section 6.

Moreover, related research is outlined in Section 7. Finally, Section 8 summarizes the contribution and provides an outlook. Subsequently, we start by giving a short introduction to Muli in Section 2, followed by a description of preliminaries of free objects (Section 2.1). All example programs presented in this paper can be compiled with the Muli compiler and executed on our modified MLVM.

2 Constraint-logic Object-oriented Programming with Muli

We base our work on the ***M**ünster **L**ogic-**I**mperative Language (Muli)*.[1] Muli is a CLOOP language whose syntax and semantics are based on those of Java 8 [DK19]. The Muli Logic Virtual Machine (MLVM) is a modification of a JVMS-compliant (cf. [Lin+15]) Java virtual machine and serves as the runtime environment. We briefly introduce the main features of the Muli programming language.

In Muli, an unbound ("free") variable is declared using the **`free`** keyword, e. g.,

```
    int width free;.
```

At runtime, free variables are treated as logic variables to be used in symbolic expressions. Free objects are declared analogously, but prior to our work their semantics was undefined and the

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

MLVM did not provide an implementation for treating free objects yet. Therefore, the following code was able to compile but the method invocation in the second line failed:

```
Shape s free;
s.getArea();.
```

This issue is tackled in the present paper, adding full support for logic variables that represent objects.

In Muli, the way all (logic and regular) variables are used in boolean or arithmetic expressions is identical to Java. However, an expression that contains unbound variables cannot be evaluated to a constant. Therefore, the MLVM treats those variables symbolically and creates a symbolic expression [DK18].[2] To give an example, after executing the Muli program in Listing 2, `five` holds the constant value `5`, whereas `symbolic` holds the symbolic expression `x + 5`.

```
int x free;
int two = 2, three = 3;
int five = three + two;
int symbolic = x + five;
```

Listing 2: Example that demonstrates symbolic evaluation of expressions that contain logic variables.

Ultimately, symbolic arithmetic expressions can evaluate to numeric constants (e. g., after substituting all contained symbolic variables by appropriate constants). For instance, an arithmetic expression that contains only **`int`** (logic) variables and **`int`** constants can be used anywhere where an **`int`** expression is expected. Therefore, symbolic expressions can be passed as parameter values, assigned to variables, or used as the return value of a method. A symbolic expression is preserved until all comprised logic variables can be substituted by a constant, either via sufficiently specific constraints or via labelling.

The behaviour described so far is deterministic. However, as soon as a symbolic expression is used as part of a condition that leads to branching (e. g., in an **`if`** statement), it is possible that the execution environment cannot decide on a unique outcome because, given appropriate constraints, a condition could be evaluated to both, **`false`** and **`true`**. For example, in the context of Listing 2 we could add

```
if (symbolic > 5),
```

which is **`true`** iff $x \geq 1$, and **`false`** otherwise. Since `symbolic` is free, both alternatives are equally possible.

Whenever more than one choice is applicable, the MLVM searches over all possible branches [DK18]. The MLVM non-deterministically selects a branch that implies a specific outcome (e. g., the condition shall be **`false`**). The resulting constraint is imposed on a constraint store that the MLVM maintains as part of its execution state [DK19]. After executing that branch, the MLVM backtracks execution state (constraint store, operand and frame stacks, program counter, and heap values) to the point where a choice was made, and then proceeds with the next branch.

With the purpose of limiting the effects of non-deterministic execution, non-deterministic branching is encapsulated. Non-deterministic programs, or *search regions*, are written in lambda expressions or methods and are passed as a parameter to one of Muli's encapsulation methods (e. g., **`getAllSolutions`**`()` or **`muli`**`()`). The result of an execution branch, i. e., either the final return value or an uncaught exception, becomes a solution to a CLOOP. The encapsulation method collects all solutions and returns them to the calling, deterministic program. Depending on the chosen encapsulation method, the surrounding program can process solutions from an array or

[2]In contrast, arithmetic expressions in Java are immediately evaluated to a constant result, i. e., the original expression is lost immediately after its evaluation. This also happens in Muli for expressions that are constant.

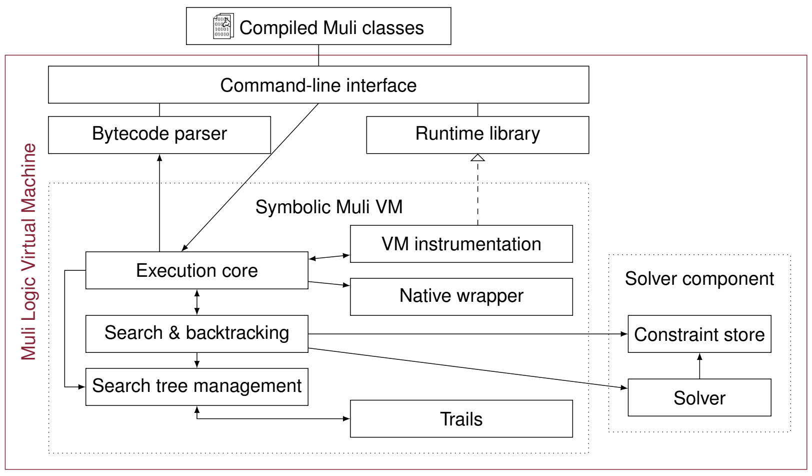


Figure 2: Conceptual structure of the MLVM. Adapted from [DK19] and updated in order to reflect recent developments.

from a stream that is evaluated non-strictly, i. e., individual solutions are computed and returned on an on-demand basis.

These features are implemented in the MLVM, whose main components are depicted in Figure 2. The execution core is a JVMS-compliant custom Java virtual machine [Lin+15] with modifications for symbolic execution of Java bytecode and encapsulated non-deterministic search [DK19]. The search tree management component maintains a search tree that represents the non-deterministic execution of search regions, where inner nodes represent non-deterministic choices and each leaf corresponds to a solution or an explicit failure. Within the execution core, the evaluation of a bytecode instruction that has non-deterministic behaviour in a search region results in the creation of a representation of the choice and its alternatives. This choice is then passed to the search tree management component, thus updating the search tree. The decision about which alternative to follow is delegated to the search & backtracking component, which imposes a corresponding constraint on the constraint store and checks whether the resulting constraint system is still satisfiable using the solver component [DK19]. The solver component currently leverages the finite domain solver JaCoP [Kuc03]; alternative solvers can easily be integrated. Once an alternative is selected, the execution core continues execution on the corresponding path. The search tree management component also maintains a set of trails that record side effects during execution. The trails are used during backtracking in order to revert side effects so that a virtual machine state is achieved that is consistent for subsequent evaluations.

2.1 Setting the Stage for Free Objects

Reference types As Muli is based on Java, Muli distinguishes the same four distinct kinds of reference types as Java does [Gos+15, § 4.3]: type variables, array types, interface types, and class types. In this work we focus on class and interface types. Subsequently, they are subsumed under *reference types* and their instances are objects, where our focus is on *free objects* in particular. Regarding classes and interfaces, the language C# has a definition of reference types that is congruent with Java's definition [Mic20]. Therefore, even though the considerations in this paper are focused on Muli, they are also applicable to other constraint-logic object-oriented programming languages, e. g., languages based on C#.

Due to the nature of Java (and, therefore, Muli), reference types are not limited to data encapsulation. With the concept of methods, class types and interface types notably encapsulate behaviour. As a consequence of method overriding and runtime polymorphism, the behaviour may also change along the implementation hierarchy. Recall the object-oriented representation of shapes from Figure 1 which will serve as our running example. The `Shape` interface prescribes subtypes to implement an appropriate method `getArea()` that calculates the area from relevant field values. Moreover, in Muli, all classes inherit from `java.lang.Object` implicitly as they do in Java [Gos+15, § 4.3.2]. For the purpose of the running example, assume that each shape also overrides the default implementations of `toString()` and `equals()` that were inherited from `java.lang.Object`, facilitating representations of field values in a human-readable form as well as comparisons with other objects.

As a result, when a variable is declared, e. g., `Shape s free`, `s` can in fact hold an instance of any class that implements `Shape`. If `Shape` were a non-abstract class, an instance of `Shape` would be a possible object as well. The fact that the actual type potentially differs from the declared type affects the type casts that can (validly) be performed on `s` at runtime, as well as the behaviour that is expected from invoking methods on the object. Consequently, adding support for free objects requires

- a non-arithmetic constraint that enforces the type (or, rather, a set of possible types) of a free object, and
- a way to discover the implementation hierarchy from all available classes.

Class discovery In order to tackle the latter, we need to make a decision regarding which classes are considered. In Java and Muli programs, classes may be available to an application even though they are not (yet) in memory from the start of an application. Instead, they reside as `.class` files in a pre-defined location on disk (the so-called class path) and are loaded on-demand by the class loader [Lin+15, § 5.3.5]. As a consequence, we can decide whether only those classes are considered that have already been loaded, as opposed to taking all classes into consideration that are on the class path. The first alternative implies that a fresh program such as

```
Shape s free;
```

might not find any implementations for `s`, unless classes that implement `Shape` were actively loaded, e. g., by constructing dummy objects from relevant classes as in

```
new Rectangle(); new Cuboid(); new Cube(); new Square();
Shape s free;.
```

Since a necessity for creating dummy objects creates additional mental load for developers, we instead propose to consider all available classes on the class path, at the cost of additional overhead for discovering and parsing all classes that are on the class path. In that case,

```
Shape s free;
```

is sufficient to instantiate a free object that can be specialized to any of its subtypes. Performing ex-ante class discovery imposes a limitation, namely that we operate under a closed-world assumption and only take classes into consideration that are present on disk at the start of the application. In Java, applications are able to create and load additional classes on the fly. However, as this feature is used rarely, it is hardly a practical issue that these classes would not be discovered.

Instances of free objects With a declaration `C o free;`, `o` becomes a logic variable that could, in theory, be substituted for either of the following:

1. an existing object from memory (the heap) that is type-compatible with `C`, thus re-using existing objects from different contexts;
2. a fully-generated fresh object, making a non-deterministic choice to branch non-deterministically over all possible alternative instances at the time of declaration; or
3. a fresh symbolic object `o`, which is further specified on an on-demand basis by imposing constraints on `o`.

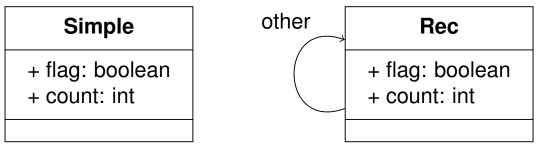


Figure 3: Class definitions. With a recursive definition, e. g., for `Rec`, exhaustive generation of concrete objects does not terminate.

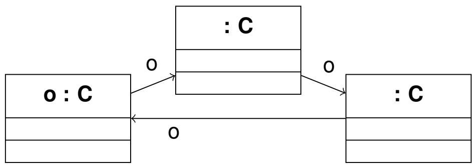


Figure 4: Example for an object structure that forms a ring.

In other contexts, a declaration `C p` implies that `p` is a fresh object, unless an existing object is assigned to `p` explicitly. As a consequence, we consider the first alternative semantically dangerous. Moreover, if existing objects were re-used, we would need to ensure that their respective scopes are not violated.

The second alternative is problematic as well since we would need to generate concrete object instances, or rather, full(!) object graphs: For an arbitrary class `C` in `C o` **`free;`**, the number of possible instances for which `o` can be substituted can be very large or even infinite. Firstly, because there may be (finitely) many specializations for `C`. Secondly, because the definition of `C` (or of a subtype) may be recursive, in turn containing a field of type `C` (or a subtype).[3] For instance, consider the definitions of `Simple` and `Rec` presented in Figure 3. The set of possible instances for objects of type `Simple` is a combination of the field values: $\{(true, -2147483648),\ (false, -2147483648),\ (true, -2147483647),\ (false, -2147483647),\ \ldots\}$. Therefore, with 2^{33} possible instances, the set is already very large to branch over, even though the class definition is relatively simple. Consequently, the state space becomes very large if we generate objects and branch non-deterministically at the point of declaration. The situation becomes even worse given a recursive class definition such as `Rec`, for which the generation of an instance does not terminate unless, at some level, `other ==` **`null`**. For the same reason, exhaustively generating *all* instances is impossible.

For these reasons we resort to the third alternative, i. e., the declaration merely constructs a symbolic variable that can be substituted for a fresh object which is not yet known. For instance, with the definitions from Figure 3 and a closed-world assumption, an object `Rec s` **`free;`** is sufficiently specific since there are no subtypes of `Rec`, and for a symbolic variable it is possible to ignore the recursive type definition. The free object can then be made more specific by using it, e. g., by invoking a method (see Section 3) or by adding constraints over its fields (see Section 4).

Since we do not re-use existing objects, free objects are always fresh instead of picking applicable instances from memory. Consider a method `isRing(C o)` as displayed in Listing 3. As a consequence of excluding re-use, an invocation `C o` **`free;`** `isRing(o)`, cannot construct an object graph as illustrated in Figure 4, since each free object of type `C` would only generate another fresh free object of type `C` as its field, without being able to refer to existing objects from the heap. Therefore, generating a ring structure by chance is not possible.

Nevertheless, we can still use non-deterministic search to generate ring structures of arbitrary lengths intentionally. This is achieved with a search region that closes a ring structure by explicit assignment, as demonstrated in Listing 4. Using non-deterministic choice over a free boolean variable, the search region either grows the ring by instantiating a fresh free object and assigning it as the next element, or closes it by assigning the first element of the ring as the next.

[3]Alternatively, the same situation occurs if `C` contains a field of a type that has a recursive definition.

```
public boolean isRing(C o) {
  Set<C> seen = new HashSet<>();
  while (o != null) {
    if (seen.contains(o)) {
      return true;
    } else {
      seen.add(o);
      o = o.o; } }
  return false; }
```

Listing 3: A method that checks whether an object `o` contains a ring structure, such as the one from Figure 4.

```
public static void main(String[] args) {
  Stream<Solution<C>> rings = Muli.muli(() -> {
    C o free; return generateRing(o, o); }); }
public C generateRing(C first, C o) {
  boolean closeRing free;
  if (closeRing) {
    o.o = first;
    return first;
  } else {
    C next free;
    o.o = next;
    return generateRing(first, o.o); } }
```

Listing 4: Using non-deterministic choice for generating ring structures of arbitrary length, such as the one in Figure 4.

Instantiating a free object Since free objects are treated symbolically, a lot of instantiation effort is skipped. Specifically, constructors that would normally initialize an object are ignored. This is necessary for two reasons: first, because the MLVM only has partial information about the actual type; and second, because a class may present multiple constructors (and might even block the default constructor). Instead, since at least the supertype is known, the MLVM can initialize fields according to the definition of the supertype by putting appropriate free variables into every field. For example, a free object of type `Rec` (Figure 3) is initialized with `{flag = boolean free, count = int free, other = Rec free}`. However, there is one notable exception: We want free objects of a type to be consistent with regular objects of the same type. This implies that *static* fields of a type need to hold the same values for all objects of that type, regardless of whether an object is free. Therefore, if a free object is the first of its type to be instantiated, we do call static initializers of the type to ensure consistency with objects that are created later. If it is not the first, it will consistently use the same static values as other objects of its type.

Now that we have clarified what free objects look like, we continue by discussing specific interactions with free objects. Section 3 tackles method invocation, whereas Section 4 presents field access. In addition, Section 5.1 discusses operations that work explicitly on types, and Section 5.2 explains the notion of equality against the presence of both, free objects and regular objects.

```
public static void main(String[] args) {
  List<Solution<String>> solutions = Muli.getAllSolutions( () -> {
    Shape s free;
    if (s.getArea() == 16)
      return s.getClass().getName();
    else
      Muli.fail(); }); }
```

Listing 5: A search region that branches over the types of a free object and returns the selected classes' names.

3 Method Invocations on Free Objects

Recall the example structure from Figure 1, in which `Shape s free;` instantiates a free object that can, in fact, assume one of four distinct actual types. Its actual type is irrelevant, unless the free object is used. Therefore, once we attempt to invoke a method that is defined in `Shape`, e. g., `getArea()`, the type becomes important since the behaviour changes depending on the type. Moreover, given a single free object, using `getArea()` from one implementation and `toString()` from another would result in inconsistent behaviour and therefore does not make sense. As a consequence, when we select an implementation for invocation, we commit the free object to the type that corresponds to the selected implementation. All implementations are equally possible, so choosing is non-deterministic.

For instance, consider the program presented in Listing 5. It searches for instances of `Shape s` whose area (as determined by `getArea()`) is 16, and then returns the name of the actual class whose implementation has been selected. Consequently, given the structure in Figure 1, we expect a solution array containing the following strings (in any order): `{"Rectangle", "Square", "Cuboid", "Cube"}`.

The program contains two method invocations on `s` that are discussed in the following. The first invocation is to `s.getArea()` on an unbound `s`. Non-deterministically selecting and invoking an implementation commits `s` to a specific type, thus binding `s`. Therefore, on the second invocation to `s.getClass()`, `s` is sufficiently specific, so that only a unique implementation of `getClass()` is possible, namely, the one that a class inherits from `java.lang.Object`. Consequently, the second invocation is deterministic. Similarly, an invocation `s.toString()` would be deterministic as well, even though every class provides its own implementation: Resulting from the binding that occurs when `s.getArea()` is called, the type of `s` is sufficiently specific so that only one `toString()` implementation can possibly be selected while maintaining consistency with previous behaviour.

Generalizing from this example, for a given object `o` on which an invocation `o.m()` is performed, the runtime environment needs to discover the set of types that provide an implementation for `m()`. This discovery needs to take into consideration which types `o` may assume. Algorithm 1 calculates the set of possible implementations for `m()`, as explained subsequently. For non-free objects `o` whose class has a definition for `m()`, the returned set is a singleton (or empty in the invalid case that the type of `o` does not provide an implementation, thus yielding a runtime exception). Therefore, invocation is deterministic. For free objects, the returned set may have more elements. In that case, invocation results in a non-deterministic choice.

Methods in Java and Muli can be overloaded, so in order to target a specific overloading, `Method` signifies a combination of a method name and its descriptor, i. e., parameter types and return type [Lin+15, § 4.3.3]. In the beginning, Algorithm 1 initializes an empty set *impls* that will later contain the invocation candidates. Afterwards, `jvmsLookup()` looks up a method implementation

Algorithm 1: Discovering the set of method implementations that are candidates for invocation.

```
implementations(Object target, Method m)
    Let $impls$ := { };
    Method $mostSpecificFromSupertypes$ := jvmsLookup ($m$, $target$.class);
    if $mostSpecificFromSupertypes$ != null then
        $impls$ += $mostSpecificFromSupertypes$;
    Let $types$ := $target$.getPossibleTypes ();
    foreach type ∈ types do
        Method $implementation$ := $type$.getMethod ($m$);
        if implementation != null && !implementation.isAccAbstract () then
            if type.isAccAbstract () —— type.isAccInterface () then
                Let $subtypes$ := $type$.getImmediateInstantiableSubtypes ();
                foreach subtype ∈ subtypes do
                    $impls$ += $subtype$.getMethod ($m$);
            else
                $impls$ += $implementation$;
    return $impls$;
```

upwards along the class hierarchy using the known lookup procedures defined in the JVMS [Lin+15, § 6.5 (*invokeinterface* and *invokevirtual*)]. This implementation is the one that will be invoked if the free object assumes its supertype ($target$.class). Afterwards, `implementations` looks at each type that the free object may assume ($target$.`getPossibleTypes()`), searching for individual implementations (`getMethod()`). If the type that contains a found implementation is not marked as abstract or as an interface, its implementation is directly added to *impls*. Otherwise, methods from the type's immediate, instantiable subtypes[4] are added to *impls* as the original type could not be instantiated. Finally, *impls* is returned to the MLVM and is used to create the non-deterministic choice for invocation.

When an implementation alternative is selected, the runtime environment has to add a constraint $types(o) = T$ to the constraint store before executing a specific method body, where the set of types T depends on the selected implementation alternative. This constraint is added in order to ensure that, after the MLVM chooses an implementation alternative, it commits to that choice regarding later interactions with the object `o`, thus narrowing its type.

To explain the construction of the set T, have a look at the artificial implementation hierarchy displayed in Figure 5: There is a class `A` that implements a method `m()`. `B` inherits from `A` and overrides `m()`, adding custom behaviour. In contrast, `C` inherits from `B` but does not add custom behaviour. Last but not least, `D` inherits from `C` and provides an implementation for `m()`. Now, for a free object `A a` **`free`**, $S = \{\texttt{A}, \texttt{B}, \texttt{C}, \texttt{D}\}$ contains the possible instance types. On invocation of `a.m()`, Algorithm 1 discovers the implementations provided by `A`, `B`, and `D`. After selecting one of the implementations, the actual type of `a` can still be one from a set of types. Specifically, the type of `a` can either be the type that provides the implementation or one of its subtypes, *except for* subtypes that provide their own implementation (as *their* respective implementation would have needed to be invoked otherwise). We call this reduced set of types T. Exemplarily, this is illustrated in Figure 5 where the possible types are constrained to $T = \{\texttt{B}, \texttt{C}\}$ after selecting the implementation `B.m()`. Even though `D` also is a subtype of `B`, it is not part of T as it provides an own implementation of `m()` and would therefore conflict with having chosen `B`'s implementation.

Furthermore, for the sake of completeness, assume that `B`'s implementation of `m()` calls a method `n()`, for which `C` provides its own implementation. Choosing an implementation for `m()` still

[4]That is, direct subtypes that are not abstract or, otherwise, their immediate instantiable subtypes.

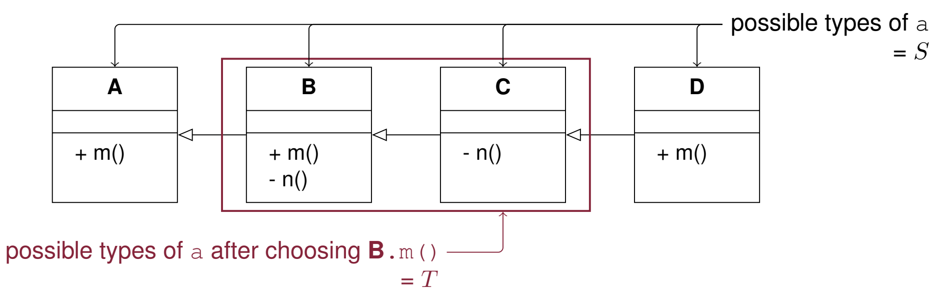

possible types of a = S
A
+ m()
B
+ m()
- n()
C
- n()
D
+ m()
possible types of a after choosing **B**.m() = T

Figure 5: Applicable instance types for a given object `A a free` before and after choosing a specific implementation.

```
class Supertype { public int field = 2; }
class Subtype extends Supertype { public int field = 1; }
class FieldShadowing {
  public static void main(String[] args) {
    Supertype a = new Supertype();
    Supertype b = new Subtype();
    a.field == b.field; // This is true!
  } }
```

Listing 6: Subclasses can hide fields of their supertypes, but fields are never overridden.

reduces the set of types to `B` *and* `C`, but the later invocation of `n()` from within `m()` results in further non-deterministic branching over the two types and their respective implementations.

In Appendix A we present a modification of Muli's operational semantics, esp. Equation (Invoke-ND), that incorporates non-deterministic choice in method invocation. The MLVM implements Algorithm 1 and represents the non-deterministic choice by creating a `Choice` node in the active search tree. The newly created `Choice` node has one branch for each invocable implementation alternative. For each branch, an appropriate type constraint is declared that maintains integrity regarding the selected implementation. When the MLVM selects an alternative from the `Choice` node, the corresponding constraint is imposed by adding it to the constraint store, and the constraint is removed again before an alternative is evaluated.

4 Field Access on Free Objects

As in Java, fields of an object are accessed in Muli using a dot notation, e. g., `square.width` given an object `Square square`. In an implementation hierarchy, fields are special in that subclasses cannot override fields that they inherit. For instance, consider the example presented in Listing 6. The subclass declares a field with a name that is identical to that of a field in the superclass, and assigns a different value. However, as a result, the original field is merely hidden from the subclass, i. e., instances of the subclass actually have two fields with the same name [Gos+15, §§ 8.3 and 9.3]. In the example of Listing 6, this implies that the condition `a.field == b.field` is true, even though `b` is an instance of `Subtype`. Since both instances `a` and `b` are accessed through the `Supertype` type, the field `field` from the superclass is used in both accesses.

The example from Listing 6 is limited to non-free objects. But as the implementation hierarchy is irrelevant for accesses to fields of regular objects, accessing fields of free objects also does not need to consider all possible types of an instance. Therefore, accessing a field of objects, free or non-free, is a deterministic operation that only depends on the type of the variable through which an instance is accessed.

Following the considerations regarding the initialization of free objects from Section 2.1, field access to a fresh free object yields an appropriate free variable, unless the accessed field is static. So for instance, assuming the class structure from Figure 1,

```
    Rectangle r free;
    return r.width;
```

returns a free variable of type **`int`**.

5 Other Operations on Free Objects

Method invocation and field access constitute basic functionality in object-oriented programs. However, there are further operations on objects that are affected by the introduction of free objects. In the following, we present the handling of explicit operations on the type of free objects, followed by a short discussion of the equality of (free) objects.

5.1 Type Operations

The handling of type operations is interesting for free objects as there is only partial information about their types. In Java/Muli bytecode, this affects the implementation of type checks (`instanceof` instruction) and type casts (`checkcast` instruction) [Lin+15, § 6.5]. These two instructions differ in that `instanceof` returns the result of the type check as a boolean value, whereas `checkcast` throws an exception if the (free) object on the operand stack cannot be cast to the intended type (otherwise, `checkcast` does nothing). For free objects, these operations are evaluated non-deterministically if either outcome is equally possible. For instance, consider the snippet in Listing 7, in which both type operations are used. When the condition is evaluated, two cases are possible: Either `instanceof` returns **`true`** if $types(s) = \{\texttt{Rectangle}, \texttt{Cuboid}\}$ holds, or **`false`** otherwise. The cast in the **`true`** branch is always possible since the possible types of `s` are sufficiently constrained as a result of evaluating `instanceof`. Therefore, the cast operation is deterministic in this example.

```
Shape s free;
if (s instanceof Rectangle) {
  Rectangle r = (Rectangle) s;
  r.width = r.height; }
```

Listing 7: Using type operations on a free object.

Generally, whether type operations on a free object are non-deterministic depends on the type constraints that are imposed on the object. A type operation `o` **`instanceof`** `T` or `(T)o` involves a free object o and a target type T. For the decision whether an operation is non-deterministic, we define the set $SuccessfulTypes_{o,T}$ that contains all types that o may assume and that are also subtypes of or equal to T (with $a \preceq b$ meaning that a is a subtype of b or b itself):

$$SuccessfulTypes_{o,T} = \{t | t \in types(o), t \preceq T\}$$

If o is of a type $\in SuccessfulTypes_{o,T}$, the type operation is successful w. r. t. the Java Virtual Machine Specification [Lin+15, § 6.5]. An additional set, $AdverseTypes_{o,T}$, contains all types for

```
Shape s free;
Rectangle r = new Rectangle();
r.width = 100; r.height = 101;
return s.equals(r); // Variation 1,
// or...
return r.equals(s); // Variation 2.
```

Listing 8: Example program involving non-determinism in the check for value equality.

which the operation would fail if o assumes one of these types:

$$AdverseTypes_{o,T} = \{t | t \in types(o), t \npreceq T\}$$

In relation to the set of currently possible types $types(o)$ that the object o may assume, the two sets are disjoint and their union comprises all possible types. $SuccessfulTypes_{o,T}$ and $AdverseTypes_{o,T}$ can be used for determining how both type operations are evaluated: If there is at least one element in $SuccessfulTypes_{o,T}$ the implication is that the type operation can be successful. Similarly, if $AdverseTypes_{o,T}$ is not empty, the type operation can fail. These two cases are not mutually exclusive; if both sets contain at least one element, execution branches non-deterministically.

Similar to how invocation is implemented in the MLVM, this implies that, at runtime, a `Choice` node is created, containing the two alternatives as branches with appropriate type constraints that make use of the sets calculated previously. For the branch that represents a successful type operation on a free object o, the MLVM imposes a constraint $types(o) = SuccessfulTypes_{o,T}$. Similarly, for the alternative branch, $types(o) = AdverseTypes_{o,T}$. Before continuing execution with one of the branches of a non-deterministically evaluated type operation, the MLVM imposes the corresponding constraint, thus ensuring that the assumptions regarding a free object are consistent within a branch of execution.

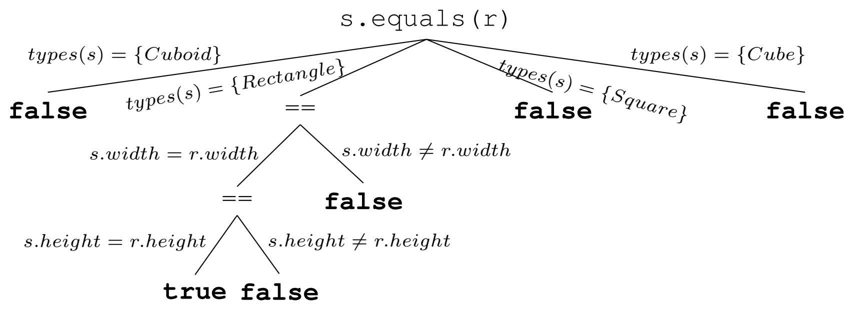
5.2 Equality

In Java, and therefore in Muli, there are two distinct notions of equality of objects [Gos+15, § 15.21.3]. *Reference equality* is different from *value equality* in that reference equality compares the addresses of two objects, but not their contents. Therefore, reference equality of two objects implies that they are, in fact, the same. Reference equality is tested using the == or != operators. In contrast, value equality between two objects is tested by invoking the `equals()` method on one object, passing the other as an argument.[5] Classes may override `equals()` in order to determine whether two objects are value-equal, thus giving developers the opportunity to decide which fields must be identical for two objects to be considered equal (if any).

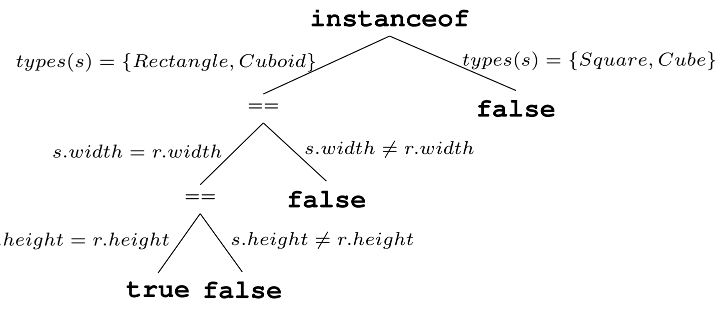
In the context of free objects, we do not need to make any particular considerations on how to handle equality for two reasons. First, using == or != on objects deterministically compares the addresses, where free objects make no exception. Second, invoking an `equals()` method on a free object will cause the MLVM to non-deterministically evaluate the method invocation. Since `equals()` does not differ from other methods, no special handling is required other than what we discussed w. r. t. method invocation (see Section 3).

For the purpose of illustration, consider Listing 8 in the context of our running example. The code uses two variations for comparing the same two objects w. r. t. value equality. Variation 1

[5]The default implementation provided by the `Object` class falls back to comparing reference equality.



(a) Variation 1



(b) Variation 2

Figure 6: Execution trees created as a result of calling `equals()` on free (Variation 1) or non-free (Variation 2) objects.

invokes the `equals()` method on the free object `s`. In contrast, Variation 2 invokes that method on a specific object, but passes `s` as a parameter. The resulting execution flows are illustrated in Figure 6. Variation 1 branches immediately on invocation of `equals()`, thus creating one branch per implementation of `equals()`, whereas the invocation itself is deterministic in Variation 2. In contrast, Variation 2 branches primarily because **`instanceof`** is called, checking the type of the free object (cf. Section 5.1). Both variations create comparable branches in case both the regular object and the free object are instances of `Rectangle`, which is ensured for the free object by imposing an adequate constraint. In that case, `Rectangle`'s implementation of `equals()` also compares the field values, ensuring that the custom value equality criterion is met. Furthermore, the illustration of the execution trees emphasizes that, in contrast to `==`, `equals()` is not commutative for free objects.

6 Demonstration

The shape application example has been useful for explaining the concepts introduced by free objects. We proceed by discussing two example applications in order to demonstrate that free objects improve the expressiveness of other Muli applications as well.

As the first example, consider the n-Queens problem as a classic search problem (cf. e. g., [FA03, Section 12.3]). Listing 9 presents a solution in Muli, assuming a class structure as illustrated in Figure 7. The search region initializes object representations of the board and of the n queens. Constraints are imposed by invoking `isOnBoard()` on the board object, thus restricting the positions of queens to $0 < x \leq n$ and $0 < y \leq n$ in accordance with the board size $n \times n$. Moreover, `threatens()` imposes constraints such that two queens may never share a diagonal, row, or column.[6] `Muli.`**`fail`**`()` is invoked when constraints are not fulfilled. As a result, the search

[6]For reference, the specific implementation of these two methods is displayed in Appendix B.

```
public static void main(String[] args) {
  Solution<Queen[]> solution = Muli.getOneSolution(() -> {
    final int n = 8; Board board = new Board(n);
    Queen[] qs = new Queen[n];
    for (int i = 0; i < n; i++) {
      Queen q free; qs[i] = q; }
    for (int i = 0; i < n; i++) {
      if (!board.isOnBoard(qs[i])) Muli.fail();
      for (int j = i+1; j < n; j++)
        if (board.threatens(qs[i], qs[j]))
          Muli.fail(); }
    return qs; });
  for (Queen q: solution.value)
    System.out.println("(" + q.x + "," + q.y + ")"); } }
```

Listing 9: n-Queens search region that makes use of object-oriented features for the implementation of a search problem.

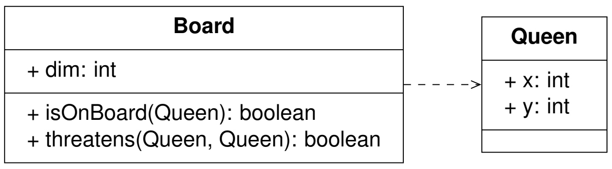

Board
+ dim: int
+ isOnBoard(Queen): boolean
+ threatens(Queen, Queen): boolean
Queen
+ x: int
+ y: int

Figure 7: Object-oriented representation of board and queens.

region only returns placements that satisfy the constraints. This example results in two interesting observations. First, CLOOP facilitates logical grouping of data and constraint definitions using classes and objects. This is illustrated by the `Board` class that stores its dimensions in a field and derives constraints accordingly, thus demonstrating encapsulation of data and behaviour in CLOOP programs. Consequently, encapsulation in classes can be used for the purpose of structuring the constraint problem, instead of using intransparent encodings that would require additional explanations. Second, we can leverage free objects for encoding the unknown state, i. e., the placement of queens on the board.

As the second example, consider an application that systematically generates directed acyclic graphs using non-deterministic search. Such an application is useful, for example, in order to use the generated graphs to describe the hidden layers of a feed-forward artifical neural network (ANN). In an ANN, the simplest structure is an empty graph, so that the ANN's input nodes are directly connected to all output nodes. Starting from the empty graph, two operations increase the size of the graph: Either adding a hidden layer (with one node as a starting point), or adding a node to one of the hidden layers. Using Muli, we can implement a search region that enumerates graph structures by non-deterministically choosing one of these operations. Additionally, for the `AddNode` operation, it non-deterministically selects the layer to which a node is added. The non-deterministic choice for one of the operations can be implemented using the equivalent of a coin flip, i. e., using a free variable **`boolean`** `coin` **`free;`** and branching over that, adding a layer if it evaluates to **`true`** and adding a node to a layer otherwise. However, the mappings of **`true`** and **`false`** require explanation. Instead, with free objects and non-deterministic choice for method invocation, we can express the non-deterministic choice for selecting an operation using a free object. Given the class structure from Figure 8, we can implement a search region that instantiates an empty graph and systematically grows it by non-deterministic choice. The search region is displayed in Listing 10. In particular, note the free variable `Operation op` **`free;`** that the runtime environment non-deterministically binds to a specific type by invoking `perform()` on it, thus choosing an operation. As an implementation detail, a third operation is added that returns the current graph structure as a solution. In contrast, the other operations perform their

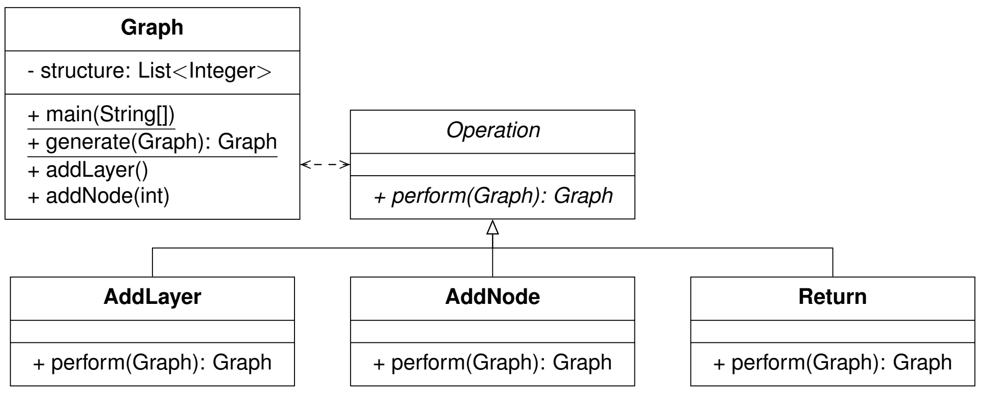


Figure 8: Representation of graph modification operations in a class structure for the purpose of non-deterministic choice.

```
public class Graph {
  // <...>
  public static void main(String[] args) {
    Stream<Solution<Graph>> graphs = Muli.muli(() -> {
      Graph basic = new Graph();
      return Graph.generate(basic); });
    // Consume graphs from the stream, e.g., for output.
  public static Graph generate(Graph g) {
    Operation op free;
    return op.perform(g); } }
```

Listing 10: Search region that generates directed acyclic graphs using non-deterministic method invocation on a free object.

modification on the `Graph` object and subsequently invoke `generate()` with the modified graph in order to proceed with the next operation. As a result, the possible variations in behaviour are encapsulated in classes that are named according to their function. Consequently, the search region itself can remain on a high abstraction level, whereas implementation details are moved into the respective classes.

Limitations This work comes with some limitations. First, it only considers objects as logic variables. Other kinds of reference types, esp. arrays, have a different structure and have therefore been left out of scope. Future work needs to tackle arrays as logic variables. Second, a potential parallelization of Muli applications is currently disregarded. The reason for that is that non-deterministic search in combination with parallelism results in state space explosion, resulting in search trees that can hardly be managed. Third, all considerations were discussed using a Java-based CLOOP language. However, they should be applicable to other (future) CLOOP languages as well because, for instance, C# uses a similar definition of reference types as that of Java.

7 Related Work

The present work is inspired by the concepts introduced in [Dag19]. In addition to the concepts discussed previously, our work is novel in that it provides a set of algorithms and an actual implementation in the context of the MLVM. As opposed to treating free objects symbolically and only resolving them as needed, other work has considered the object generation approach, particularly against the background of software test-data generation [Kor90; ZL07; DZZ08]. The authors present different ways of generating all possible object instances before they are used in a program. Also in the context of test-case generation, symbolic execution for Java objects has been explored, mutating object types in order to generate test data [LLX17]. However, their approach requires initialization of reference-type logic variables, as opposed to treating them purely symbolically. Moreover, they require bytecode to be augmented with instrumentation instructions for symbolic execution, thus requiring re-compilation even of used libraries. In contrast, the MLVM operates on standard-compliant, unmodified JVM bytecode. Moreover, special kinds of objects, namely strings and lists, have been investigated, but the results are limited to data encapsulation. Krings et al. translate string operations into constraints using a Prolog-based constraint-handling rules system [Kri+20]. Since even the resulting strings are formulated only in Prolog, the applicability in an integrated language is limited. Lists can also be treated symbolically like our free objects, initializing just as much of them as needed for a specific program that uses the lists [KPV03].

Generally, integrating features from declarative paradigms into mainstream programming languages has proven useful. Prominent examples are the integrations of concepts from functional programming, such as the Stream API for Java and LINQ for .NET, but also integrated languages such as Scala [Ode+17; Hun18]. The language Kaleidoscope'91 attempts an integration of an object-oriented language with constraints, facilitating the specification of constraints over fields of one or more objects [FB92]. This already adds a sense of declarativity that is unknown in most current object-oriented programming languages, as fields could be formulated as expressions that combine other fields. Nevertheless, the authors do not discuss objects that are completely free as in the sense of this work, i. e., whose types are only partially known. Other work leverages object-orientation capabilities by using objects for modelling declarative expressions, e. g., for integrating Prolog search into Java programs [Ost15]. Alternatively, there are constraint solver libraries for Java, such as OptaPlanner [The20] and JaCoP [Kuc03]. Neither of these approaches achieve a full integration of constraint-logic features into an object-oriented programming language and merely provide an object-oriented abstraction layer for (some) constraint-logic features.

Closely related to CLOOP is the paradigm of functional-logic programming, most prominently represented by Curry [HKM95]. Analogous to CLOOP, functional-logic programming adds features from logic programming to a functional programming language, resulting in programming languages that are similarly non-deterministic. On the one hand, Curry has algebraic data types that resemble the data-encapsulation behaviour of Muli objects. On the other hand, neither Curry nor other functional-logic programming approaches consider encapsulation of behaviour. Similarly, constructors in Prolog merely encapsulate data. Therefore, encapsulation of behaviour and its non-deterministic evaluation are novel contributions of the present work.

8 Concluding Remarks

In this paper we add the concept of free objects to constraint-logic object-oriented programming. Specifically, this work contributes the concept of logic variables with a class type or an interface type. These are particularly interesting because, in addition to encapsulating data, they also encapsulate behaviour. To that end, we propose and implement a semantics for interacting with free objects at runtime, taking non-deterministic choice over the encapsulated behaviours into account. We demonstrate our concepts by implementing them in the MLVM, i. e., in the runtime environment used by the CLOOP language Muli. A modified MLVM that includes our implementation is available on GitHub.[7]

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

We have shown that adding free objects to a constraint-logic object-oriented programming language improves the expressiveness of the language. CLOOP languages were already useful in applications that interleave imperative code with non-deterministic search. With the recent additions, CLOOP can also be used to express traditional constraint-logic problems in novel ways using object representations that also encapsulate behaviour, such as n-Queens with methods that explain constraints by using descriptive names. Moreover, CLOOP can be used for an effective formulation of new problems.

Even though the considerations in this paper are focused on Muli, they are also applicable to other constraint-logic object-oriented programming languages. For instance, since C#'s definition of reference types is congruent to that in Java, future work could port the results to a (future) constraint-logic object-oriented programming language that is based on C#. Future work sets out to add support for free arrays in order to incorporate another kind of reference type.

A Operational Semantics of Muli (Excerpt)

The operational semantics is defined for a core subset of Muli [DK18]. Here, we first display an excerpt from the reduction rules that are relevant to a rule that we modify with this paper. Subsequently, the modified rule is presented. Throughout the definitions, modifications to their respective originals that were necessary in order to add support for free objects are indicated in red.

Symbols In this reduction semantics, computations depend on an environment, a state, and a constraint store [DK18].

- $Env = (Var \cup \mathcal{M}) \to (\mathcal{A} \cup (Var^* \times Stat))$: Set of all environments, mapping variables $\in Var$ to addresses $\in \mathcal{A}$ and methods $\mathcal{M}$ to a tuple $((x_0, x_1, \ldots, x_k), s)$, signifying parameters and a code body s. Note that we add x_0 to the original definition. x_0 shall hold the object that a method was invoked on, unless the method is static.
- $\rho_0 \in Env$ is a special initial environment that maps functions to their respective parameters and code (under the simplifying assumption that classes and their methods are in global scope).
- $\Sigma = \mathcal{A} \to (\{\bot\} \cup Tree(\mathcal{A}, \mathbb{Z}))$: Set of all possible memory states.
- A special address α_0 with $\sigma(\alpha_0) = \bot$ is reserved for holding return values of method invocations.
- $CS = \{\texttt{true}\} \cup Tree(\mathcal{A}, \mathbb{Z})$: Set of all possible constraint store states.
- $\rho \in Env$, $\sigma \in \Sigma$, $\gamma \in CS$. Discriminating indices are added if necessary.
- $a[x/d]$ is used for modifications to a state or environment a, meaning

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

- The semantics of expressions is described with the infix relation

$$\to \subset (Expr \times Env \times \Sigma \times CS) \times ((\mathbb{B} \cup Tree(\mathcal{A}, \mathbb{Z})) \times \Sigma \times CS).$$

- The semantics of statements is described by the infix relation

$$\rightsquigarrow \subset (Stat \times Env \times \Sigma \times CS) \times (Env \times \Sigma \times CS).$$

Syntax The grammar is only modified slightly, incorporating method invocations on objects and reference type variables. Otherwise, taken from [DK18].

$$
\begin{array}{ll}
e ::= & c \;\mid\; x \;\mid\; e_1 \oplus e_2 \;\mid\; x.m(e_1, \ldots, e_k) \\
& \text{where } c \in \mathbb{Z},\; x \in Var,\; e_1, \ldots, e_k \in AExpr,\; \oplus \in AOp,\; k \in \mathbb{N}, \\
& x.m \text{ can be resolved to an implementation } i \in \mathcal{M}, \\[1em]
b ::= & e_1 \odot e_2 \;\mid\; b_1 \otimes b_2 \;\mid\; \texttt{true} \;\mid\; \texttt{false} \\
& \text{where } e_1, e_2 \in AExpr,\; b_1, b_2 \in BExpr,\; \odot \in ROp,\; \otimes \in BOp, \\[1em]
s ::= & ; \;\mid\; \texttt{int}\; x; \;\mid\; \texttt{int}\; x\; \texttt{free}; \;\mid\; \texttt{T}\; x; \;\mid\; \texttt{T}\; x\; \texttt{free}; \;\mid\; x = e; \;\mid\; e; \;\mid\; \{s\} \;\mid\; s_1\, s_2 \;\mid \\
& \texttt{if}\; (b)\; s_1\; \texttt{else}\; s_2 \;\mid\; \texttt{while}\; (b)\; s \;\mid\; \texttt{return}\; e; \;\mid\; \texttt{fail}; \\
& \text{where } x \in Var,\, e \in AExpr,\; b \in BExpr,\; s, s_1, s_2 \in Stat,\; \texttt{T} \text{ is a class or interface type.}
\end{array}
$$

Reduction Rules The following reproduces reduction rules as preliminaries (cf. [DK18]), before presenting a rule modification for non-deterministic invocation as required for this paper.

Variable resolution:

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

Arithmetic expressions, resulting either in a constant value if nested expressions are constant, or in a symbolic expression otherwise:

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

$$\frac{\langle e_1, \rho, \sigma, \gamma \rangle \rightarrow (v_1, \sigma_1, \gamma_1),\;\; \langle e_2, \rho, \sigma_1, \gamma_1 \rangle \rightarrow (v_2, \sigma_2, \gamma_2),\;\; \{v_1, v_2\} \not\subseteq \mathbb{Z}}{\langle e_1 \oplus e_2, \rho, \sigma, \gamma \rangle \rightarrow (\oplus(v_1, v_2), \sigma_2, \gamma_2)} \tag{AOp2}$$

Non-deterministic Invocation Under the assumption of global functions and disregarding object-oriented features, the operational semantics of invocation is defined as a deterministic operation [DK18]:

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

Adding support for object-orientation and the availability of multiple implementations for an object, we modify and replace the rule as presented subsequently (changes highlighted in red). The new rule uses the set $implementations(o, m)$ that is calculated using Algorithm 1 in order to find possible implementing types.

$$\frac{\begin{array}{r}\langle o, \rho, \sigma, \gamma \rangle \rightarrow (v_0, \sigma, \gamma),\;\; \langle e_1, \rho, \sigma, \gamma \rangle \rightarrow (v_1, \sigma_1, \gamma_1),\;\; \langle e_2, \rho, \sigma_1, \gamma_1 \rangle \rightarrow (v_2, \sigma_2, \gamma_2), \\ \ldots,\;\; \langle e_k, \rho, \sigma_{k-1}, \gamma_{k-1} \rangle \rightarrow (v_k, \sigma_k, \gamma_k),\; i \in implementations(v_0, m), \\ \rho(i) = (\bar{x}_k,\; s),\;\; \langle s, \rho_0[{}^{\bar{x}_k}/_{\bar{\alpha}_k}], \sigma_k[{}^{\bar{\alpha}_k}/_{\bar{v}_k}], \gamma_k \wedge types(o) = T \rangle \rightsquigarrow (\rho_{k+1}, \sigma_{k+1}, \gamma_{k+1}),\; \sigma_{k+1}(\alpha_0) = r\end{array}}{\langle o.m(e_1, \ldots, e_k), \rho, \sigma, \gamma \rangle \;\rightarrow (r, \sigma_{k+1}[\alpha_0/\perp], \gamma_{k+1})} \tag{Invoke-ND}$$

For non-free objects $target$ whose class has a definition for m, $implementations(target, m)$ is a singleton. Therefore, invocation is deterministic. For free objects, $implementations(target, m)$ may have more elements. In that case, the evaluation of this rule becomes non-deterministic. Moreover, note that a constraint $types(o) = T$ is added after selecting a specific implementation. T depends on the selected implementation alternative as explained in Section 3 (illustrated with Figure 5).

The new rule depends on the rule in Equation (Var) for resolving the object variable o based on the state of environment and memory, and on the rules in Equations (AOp1) and (AOp2) for substituting parameter expressions. The definition assumes that the environment ρ contains every method definition, comprising a parameter definition $\bar{x}_k$ and a body s.

B Implementation of Board and Queens

The following code implements the class structure as illustrated in Figure 7.

```
public class Board {
    final int dim;

    public Board(int dim) { this.dim = dim; }

    public boolean isOnBoard(Queen q) {
        if (q.x < 0) return false;
        if (q.x > dim-1) return false;
        if (q.y < 0) return false;
        if (q.y > dim-1) return false;
        return true; }

    public boolean threatens(Queen p, Queen q) {
        if (p.x == q.x) return true;
        if (p.y == q.y) return true;
        if (p.x - p.y == q.x - q.y) return true;
        if (p.x + p.y == q.x + q.y) return true;
        return false; } }

public class Queen { int x, y; }
```

References

[Dag19] Jan C. Dageförde. “Reference Type Logic Variables in Constraint-Logic Object-Oriented Programming”. In: *Functional and Constraint Logic Programming*. Ed. by J. Silva. Vol. 11285. Lecture Notes in Computer Science. Cham: Springer, 2019, pp. 131–144. DOI: 10.1007/978-3-030-16202-3_8.

[DK18] Jan C. Dageförde and Herbert Kuchen. “An Operational Semantics for Constraint-Logic Imperative Programming”. In: *Declarative Programming and Knowledge Management*. Ed. by Dietmar Seipel, Michael Hanus, and Salvador Abreu. Vol. 10977. Lecture Notes in Artificial Intelligence. Cham: Springer, 2018, pp. 64–80. DOI: 10.1007/978-3-030-00801-7_5.

[DK19] Jan C. Dageförde and Herbert Kuchen. “A Compiler and Virtual Machine for Constraint-logic Object-oriented Programming with Muli”. In: *Journal of Computer Languages* 53 (2019), pp. 63–78. ISSN: 2590-1184. DOI: 10.1016/j.cola.2019.05.001.

[DZZ08] A. V. Demakov, S. V. Zelenov, and S. A. Zelenova. “Using abstract models for the generation of test data with a complex structure”. In: *Programming and Computer Software* 34.6 (2008), pp. 341–350. ISSN: 03617688. DOI: 10.1134/S0361768808060054.

[FA03] Thom Frühwirth and Slim Abdennadher. *Essentials of Constraint Programming*. Berlin Heidelberg: Springer, 2003. ISBN: 978-3-642-08712-7.

[FB92] Bjorn N. Freeman-Benson and Alan Borning. “Integrating Constraints With an Object-Oriented Language”. In: *ECOOP 92*. Vol. 615. 1992, pp. 268–286. ISBN: 978-3-540-55668-8. DOI: 10.1007/BFb0053042.

[Gos+15] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java® Language Specification – Java SE 8 Edition*. 2015. URL: https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf (visited on 05/03/2019).

[HKM95] Michael Hanus, Herbert Kuchen, and Juan Jose Moreno-Navarro. “Curry: A Truly Functional Logic Language”. In: *ILPS’95 Workshop on Visions for the Future of Logic Programming* (1995), pp. 95–107.

[Hun18] John Hunt. *A Beginner’s Guide to Scala, Object Orientation and Functional Programming*. 2nd ed. Springer, 2018. ISBN: 978-3-319-75770-4.

[Kor90] Bogdan Korel. “Automated Software Test Data Generation”. In: *IEEE Transactions on Software Engineering* 16.8 (1990), pp. 870–879. ISSN: 00985589. DOI: 10.1109/32.57624.

[KPV03] Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. “Generalized Symbolic Execution for Model Checking and Testing”. In: *TACAS’03 Proceedings of the 9th international conference on tools and algorithms for the construction and analysis of systems*. 2003, pp. 553–568.

[Kri+20] Sebastian Krings, Joshua Schmidt, Patrick Skowronek, Jannik Dunkelau, and Dierk Ehmke. “Towards Constraint Logic Programming over Strings for Test Data Generation”. In: *Declarative Programming and Knowledge Management*. Vol. 12057. 2020, pp. 139–159. DOI: 10.1007/978-3-030-46714-2_10.

[Kuc03] Krzysztof Kuchcinski. “Constraints-driven scheduling and resource assignment”. In: *ACM Transactions on Design Automation of Electronic Systems* 8.3 (2003), pp. 355–383. ISSN: 1084-4309. DOI: 10.1145/785411.785416.

[Lin+15] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java® Virtual Machine Specification – Java SE 8 Edition*. 2015. URL: https://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf (visited on 05/03/2019).

[LLX17] Lian Li, Yi Lu, and Jingling Xue. “Dynamic symbolic execution for polymorphism”. In: *ACM International Conference Proceeding Series* (2017), pp. 120–130. DOI: 10.1145/3033019.3033029.

[Mic20] Microsoft. *Reference types (C# Reference)*. 2020. URL: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/reference-types (visited on 04/16/2020).

[Ode+17] Martin Odersky et al. *Scala Language Specification*. 2017. URL: http://www.scala-lang.org/files/archive/spec/2.12/ (visited on 05/03/2019).

[Ost15] Ludwig Ostermayer. “Seamless Cooperation of Java and Prolog for Rule-Based Software Development”. In: *Proceedings of RuleML 2015*. 2015. URL: http://ceur-ws.org/Vol-1417/paper2.pdf.

[The20] The OptaPlanner Team. *OptaPlanner User Guide, Version 7.32.0*. 2020.

[ZL07] Ruilian Zhao and Qing Li. “Automatic Test Generation for Dynamic Data Structures”. In: *Fifth International Conference on Software Engineering Research, Management and Applications*. 2007, pp. 545–549. DOI: 10.1109/SERA.2007.64.

Working Papers, ERCIS

Nr. 1 Becker, J.; Backhaus, K.; Grob, H. L.; Hoeren, T.; Klein, S.; Kuchen, H.; Müller-Funk, U.; Thonemann, U. W.; Vossen, G.; European Research Center for Information Systems (ERCIS). Gründungsveranstaltung Münster, 12. Oktober 2004.

Nr. 2 Teubner, R. A.: The IT21 Checkup for IT Fitness: Experiences and Empirical Evidence from 4 Years of Evaluation Practice. 2005.

Nr. 3 Teubner, R. A.; Mocker, M.: Strategic Information Planning – Insights from an Action Research Project in the Financial Services Industry. 2005.

Nr. 4 Gottfried Vossen, Stephan Hagemann: From Version 1.0 to Version 2.0: A Brief History Of the Web. 2007.

Nr. 5 Hagemann, S.; Letz, C.; Vossen, G.: Web Service Discovery – Reality Check 2.0. 2007.

Nr. 6 Teubner, R.; Mocker, M.: A Literature Overview on Strategic Information Management. 2007.

Nr. 7 Ciechanowicz, P.; Poldner, M.; Kuchen, H.: The Münster Skeleton Library Muesli – A Comprehensive Overview. 2009.

Nr. 8 Hagemann, S.; Vossen, G.: Web-Wide Application Customization: The Case of Mashups. 2010.

Nr. 9 Majchrzak, T.; Jakubiec, A.; Lablans, M.; Ükert, F.: Evaluating Mobile Ambient Assisted Living Devices and Web 2.0 Technology for a Better Social Integration. 2010.

Nr. 10 Majchrzak, T.; Kuchen, H: Muggl: The Muenster Generator of Glass-box Test Cases. 2011.

Nr. 11 Becker, J.; Beverungen, D.; Delfmann, P.; Räckers, M.: Network e-Volution. 2011.

Nr. 12 Teubner, A.; Pellengahr, A.; Mocker, M.: The IT Strategy Divide: Professional Practice and Academic Debate. 2012.

Nr. 13 Niehaves, B.; Köffer, S.; Ortbach, K.; Katschewitz, S.: Towards an IT consumerization theory: A theory and practice review. 2012

Nr. 14 Stahl, F., Schomm, F., & Vossen, G.: Marketplaces for Data: An initial Survey. 2012.

Nr. 15 Becker, J.; Matzner, M. (Eds.).: Promoting Business Process Management Excellence in Russia. 2012.

Nr. 16 Teubner, R.; Pellengahr, A.: State of and Perspectives for IS Strategy Research. 2013.

Nr. 18 Stahl, F.; Schomm, F.; Vossen, G.: The Data Marketplace Survey Revisited. 2014.

Nr. 19 Dillon, S.; Vossen, G.: SaaS Cloud Computing in Small and Medium Enterprises: A Comparison between Germany and New Zealand. 2015.

Nr. 20 Stahl, F.; Godde, A.; Hagedorn, B.; Köpcke, B.; Rehberger, M.; Vossen, G.: Implementing the WiPo Architecture. 2014.

Nr. 21 Pflanzl, N.; Bergener, K.; Stein, A.; Vossen, G.: Information Systems Freshmen Teaching: Case Experience from Day One (Pre-Version of the publication in the International Journal of Information and Operations Management Education (IJIOME)). 2014.

Nr. 22 Teubner, A.; Diederich, S.: Managerial Challenges in IT Programmes: Evidence from Multiple Case Study Research. 2015.

Nr. 23 Vomfell, L.; Stahl, F.; Schomm, F.; Vossen, G.: A Classification Framework for Data Marketplaces. 2015.

Nr. 24 Stahl, F.; Schomm, F.; Vomfell, L.; Vossen, G.: Marketplaces for Digital Data: Quo Vadis?. 2015.

Nr. 25 Caballero, R.; von Hof, V.; Montenegro, M.; Kuchen, H.: A Program Transformation for Converting Java Assertions into Control-flow Statements. 2016.

Nr. 26 Foegen, K.; von Hof, V.; Kuchen, H.: Attributed Grammars for Detecting Spring Configuration Errors. 2015.

Nr. 27 Lehmann, D.; Fekete, D.; Vossen, G.: Technology Selection for Big Data and Analytical Applications. 2016.

Nr. 28 Trautmann, H.; Vossen, G.; Homann, L.; Carnein, M.; Kraume, K.: Challenges of Data Management and Analytics in Omni-Channel CRM. 2017.

Nr. 29 Rieger, C.: A Data Model Inference Algorithm for Schemaless Process Modeling. 2016.

Nr. 30	Bünder, H: A Model-Driven Approach for Graphical User Interface Modernization Reusing Legacy Services. 2019.
Nr. 31	Stockhinger, J.; Teubner, R: How Digitalization Drives the IT/IS Strategy Agenda. 2020.