





Constraint-Logic Object-Oriented Programming with Free Arrays

Hendrik Winkelmann^(✉) , Jan C. Dageförde , and Herbert Kuchen

ERCIS, Leonardo-Campus 3, 48149 Münster, Germany
{hendrik.winkelmann,dagefoerde,kuchen}@uni-muenster.de

Abstract. Constraint-logic object-oriented programming provides a useful symbiosis between object-oriented programming and constraint-logic search. The ability to use logic variables, constraints, non-deterministic search, and object-oriented programming in an integrated way facilitates the combination of search-related program parts and other business logic in object-oriented applications. With this work we add array-typed logic variables (“free arrays”), thus completing the set of types that logic variables can assume in constraint-logic object-oriented programming. Free arrays exhibit interesting properties, such as indeterminate lengths and non-deterministic accesses to array elements.

Keywords: Constraint-logic object-oriented programming · Free arrays · Non-deterministic element access · Reference types

1 Motivation

In constraint-logic object-oriented programming (CLOOP), one of the remaining missing puzzle pieces is the ability to use logic variables in lieu of arrays. As a novel paradigm, CLOOP describes programming languages that add constraint-logic features on top of an object-oriented syntax. Most importantly, CLOOP offers logic variables, constraints, and encapsulated non-deterministic search, seamlessly integrated with features from object-oriented programming. As a blueprint for CLOOP languages, the **M**ünster **L**ogic-**I**mperative Programming Language (Muli) [2] is a Java-based language that has been successfully used in the generation of artificial neural networks [5], for search problems from the domain of logistics, and for classical search problems [4]. So far, logic variables in Muli can be used instead of variables of primitive types [4] or in place of objects [6]. Adding support for array-type logic variables is another step on the path to achieving the full potential of CLOOP. Potential opportunities are illustrated with the code snippet in Listing 1. This snippet declares a logic array *a*, i.e., an array with an indeterminate number of elements and none of the elements is bound to a specific value. Moreover, it uses logic variables as indexes for accessing array elements, resulting in non-deterministic accesses.

Prior to this work, Muli supported the use of arrays with fixed lengths and logic variables as elements. In contrast, free arrays are logic variables with an

```

int i free;
int j free;
int[] a free;
if (a[i] > a[j]) a[i] = a[j] else ...;

```

Listing 1. Snippet in which an array as well as the indexes for access are not bound.

array type that are not bound to specific values, i.e., the entire array is treated symbolically. In a free array, the individual elements as well as the number of the elements are not known. This work presents the introduction of free arrays into CLOOP and Muli. The paper starts off by providing a short introduction to the Muli programming language in Sect. 2. Afterwards, it presents the contributions of this work:

- Section 3 introduces and defines the concept of free arrays in a CLOOP language.
- Section 4 discusses how to handle non-deterministic accesses to array elements when a free variable is used as the index.
- These ideas are accompanied by the description of a prototypical implementation of free arrays in the runtime environment of Muli, specifying the handling of symbolic array expressions as well as the modified behaviour of array-related bytecode instructions (see Sect. 5).

Section 6 presents related work, followed by a short summary in Sect. 7.

2 Constraint-Logic Object-Oriented Programming with Muli

Our proposal is based on the constraint-logic object-oriented programming language Muli, which facilitates the integrated development of (business) applications that combine deterministic program logic with non-deterministic search. Muli is based on Java 8 and adds features that enable constraint-logic search [4]. A key feature of Muli is the ability to declare logic variables. Since logic variables are not *bound* to a specific value, they are called *free variables*. A free variable is declared using the **free** keyword, as shown in the following example:

```
int size free.
```

Syntactically, declaring a free integer array is valid, too:

```
int[] numbers free,
```

however, the behaviour of free arrays was not defined yet, so such a declaration resulted in an exception at runtime. In this paper this issue is addressed by an implementation of free arrays.

```

Solution<Integer>[] evens = Muli.getAllSolutions(() -> {
    int number free;
    if (number > 5) {
        throw Muli.fail();
    } else if (number < 0) {
        throw Muli.fail();
    } else {
        return number*2; } }

```

Listing 2. Search region that imposes constraints on a free variable number and returns an expression as its solution.

Following its declaration, a variable can be used in place of other (regular) variables of a compatible type, for instance as part of a condition:

```
if (size > 5)
```

As `size` is not bound, the condition can be evaluated to **true** as well as to **false**, given appropriate circumstances. Upon evaluation of that condition, the executing runtime environment non-deterministically takes a decision and imposes an appropriate constraint that supports and maintains this choice (for example, `size > 5` in order to evaluate the **true**-branch). To that end, the runtime environment leverages a constraint solver for two purposes: First, the constraint solver is queried to check whether the constraint system of an application is consistent, thus avoiding the execution of branches whose constraint system cannot be solved. Second, the constraint solver is used to find specific values for free variables that respect the imposed constraints.

Eventually, the runtime environment considers all alternative decisions. The result is a (conceptual) search tree, in which the inner nodes correspond to the points at which decisions can be taken, with one subtree per decision alternative [7]. The eventual outcomes of the execution (in particular, returned values and thrown exceptions) are the leaves of the tree. A returned value or a thrown exception is a solution of non-deterministic search. In addition, Muli provides the facility to explicitly cut execution branches that are not of interest by invoking the `Muli.fail()` method.

The execution behaviour of Muli applications is, for the most part, deterministic and additionally provides *encapsulated search*. Application parts that are intended to perform non-deterministic search need to be declared explicitly in the form of methods or lambda expressions. These parts are called *search regions*. In order to start search, a search region is passed to an encapsulated search operator (e.g., `Muli.getAllSolutions()`) that causes the runtime to perform search while collecting all found solutions. After execution finishes, the collected solutions are returned to the invoking (deterministic) part of the application. Exemplarily, consider the search region presented in Listing 2. For a logic variable `number` it imposes constraints s.t. $0 \leq \text{number} \leq 5$ by cutting execution branches that do not satisfy this constraint. Otherwise, the symbolic expres-

```

ArrayList<Integer> list = new ArrayList<>();
for (int i = 0; i < (int)(Math.random()*1000); i++)
    list.add(i);
int[] arr = new int[list.size()];

```

Listing 3. The length of an array is not necessarily known at compile time. This example snippet determines the length at runtime instead.

sion number*2 is returned and collected by `Muli.getAllSolutions()`, i.e., the presented search region returns the numbers {0, 2, 4, 6, 8, 10}.

Muli applications are executed on the Münster Logic Virtual Machine (MLVM) [4]. The MLVM is a custom Java virtual machine with support for symbolic execution of Java/Muli bytecode and non-deterministic execution of search regions. The MLVM represents non-deterministic execution in a search tree, in which the inner nodes are Choice nodes (with one subtree per alternative decision that can be taken) and the leaf nodes are outcomes of search, i.e., solutions or failures [7]. Executing a bytecode instruction with non-deterministic behaviour results in the creation of a Choice node that is added to the search tree. For example, executing an `If_icmpeq` instruction (that corresponds to evaluating an equality expression as part of an `if` condition) results in the creation of a Choice node with two subtrees, one per alternative outcome, provided that the result of `If_icmpeq` can take either value according to the constraints that have already been imposed.

3 Arrays as Logic Variables

Muli relies on the symbolic execution of Java/Muli bytecode, i.e., symbolic expressions are generated during the evaluation of expressions that cannot (yet) be evaluated to a single constant value. Therefore, adding support for free arrays implies introducing symbolic arrays into the execution core of the MLVM.

The length of arrays in Java (and, therefore, in Muli) does not need to be known at compile time, as the legal code example in Listing 3 demonstrates: The number of elements that the array `arr` holds will become known at runtime. The length is arbitrary, provided that it can be represented by a positive (signed) `int` value [13].¹ As a consequence, a free array that is declared using

```
T[] arr free
```

comprises

- an unknown number of elements, so that `arr.length` is a free `int` variable n , where $0 \leq n \leq \text{Integer.MAX_VALUE}$, and
- one free variable of type `T` per element `arr[i]` with $0 \leq i < \text{arr.length}$

¹ At least in theory, as the `Newarray` bytecode instruction takes an `int` value for the length. In practice, the actual maximum number of elements may be lower as it depends on the available heap size on the executing machine.

Treating the length of a free array `arr` as a free variable provides the benefit that the length can be influenced by imposing constraints over `arr.length`, i.e., by referring to the length as part of **if** conditions. Moreover, for an array `T[] arr` **free** the type of the individual array elements `arr[i]` depends on what `T` is:

- If `T` is a primitive type, each element is a simple free variable of that type.
- If `T` is an array type, the definition becomes recursive as each element is, in turn, a free array.
- If `T` is a class or interface type, each element is a free object. Therefore, the actual type `T'` of each element is $T' \preceq T$, i.e., an element's type is either `T` or a type that extends or implements `T`. We do not go into specifics on free objects, as they are not of particular relevance here. The interested reader is directed to [6] on that matter.

Java requires regular arrays to be initialized either using an array creation expression of the form `T[] arr = new T[n];`, resulting in an array `arr` with `n` elements of type `T` [10, § 15.10.1]; or an array initializer such as `int[] arr = {1, 2};`, resulting in an integer array that holds exactly the specified elements [10, § 10.6]. For free arrays, this opens up alternative ways of declaring (and initializing) a free array in Muli.

Simple free variable declaration. First, following the syntax that is used to declare any free variable, `T[] arr` **free** declares a free array whose length and elements are indeterminate.

Modified array creation expression. Second, `T[] arr = new T[n] free;` is a modified array creation expression that allows to specify a fixed length for the array (unless `n` is free) while refraining from defining any of the array elements.

Modified array initializer. Third, a modification of the array initializer expression facilitates specifying the length as well as some array elements that shall be free; e.g., `int[] a = {1, free, 0};` would define an array with a fixed length with two constant elements and a free one at `a[1]`. Trivially, regardless of the chosen initializer, array elements can be modified after the array has been initialized using explicit assignment. For example, `a[1] = 2;` can be used to replace an element (for example, a free variable) with a constant, and `int i free; a[1] = i;` replaces the element at index 1 with a free **int** variable.

These considerations facilitate the initialization and subsequent use of logic variables that represent arrays or array elements. All three alternatives are useful and should therefore be syntactically valid. For example, Listing 4 combines the initialization of a free string array via a simple free variable declaration, followed by imposing a constraint over the array's length (with `Muli.fail()` effectively cutting the branch of execution in which that constraint would not be satisfied).

```
String[] outputLines free;
if (outputLines.length > 5) {
    throw Muli.fail();
} else {
    // <...>
}
```

Listing 4. Limiting a free array’s length to at most five elements by imposing an appropriate constraint.

4 Non-deterministic Access to Array Elements

Reconsider the example snippet from a search region that is given in Listing 1: Free arrays become particularly interesting when array elements are accessed without specifying the exact index, i.e., with the index as a free variable (e.g., `arr[i]` where `int i free`). In the comparison `a[i] > a[j]`, the array `a` as well as the indexes for access are free variables. For a more complex example, consider the application depicted in Listing 5. It shows a simple sorting algorithm. The algorithm is not particularly efficient, but rather serves to show how free arrays can be used in a Muli application and demonstrates the use of other Muli features as well. The general idea of Listing 5 is to find a permutation of the elements of `b` that leads to a sorted array `a`. In line 3 of Listing 5, a free array of indexes is introduced. In lines 8–10, the unbound elements of this array are used as indexes of the arrays `usedIdx` and `a`. The algorithm searches for a permutation s.t. the final array is sorted. Consequently, if an index is used more than once, the array `idx` does not represent a permutation and the current branch of the search fails (line 8). Then, another branch is tried after backtracking. If the considered permutation does not lead to a sorted array, the current branch of the search also fails, thus resulting in backtracking (line 12). The efficiency of the algorithm ultimately depends on the constraint solver on which the Muli runtime system relies. Currently, Muli offers using either JaCoP [12], Z3 [15] or a custom SMT solver from the Münster Generator for Glass-box Test Cases (Muggl [8]). Moreover, the MLVM provides a flexible solver component that facilitates the addition of alternative constraint solvers to the MLVM [3].

Accessing an array with a free index is a non-deterministic operation, because more than one array element (or even a thrown runtime exception) could be the result of the access operation. Subsequently, we present approaches that can be used for handling such non-deterministic accesses to arrays. This list of approaches is probably non-exhaustive as there may be additional alternatives.

A first and simple approach would be to branch over all possible values for the index `i` in case that there is an access to `a[i]` where `i free`. Effectively, this is the equivalent of a labeling operation, successively considering every array element as the result of the access. Clearly, this would lead to a huge search space and it is hence not a reasonable option in most cases.

```

1  public class SimpleSort {
2      public static int[] sort(int[] b) {
3          int[] idx free;
4          boolean usedIdx[] free;
5          int[] a free;
6          if (a.length != b.length) throw Muli.fail();
7          for (int i = 0; i < b.length; i++) {
8              if (usedIdx[idx[i]]) throw Muli.fail();
9              a[idx[i]] = b[i];
10             usedIdx[idx[i]] = true; }
11         for (int i = 0; i < b.length-1; i++) {
12             if (a[i] > a[i + 1]) throw Muli.fail(); }
13         return a; }
14     public static void main(String[] args) {
15         int[] b = {1, 42, 17, 56, 78, 5, 27, 39, 12, 8};
16         Solution<Object> result =
17             Muli.getOneSolution(() -> sort(b));
18     } }

```

Listing 5. Simple sorting algorithm that leverages free arrays.

A second approach could store constraints that involve accesses to array elements with unbound indexes symbolically. In the example from Listing 1, this implies storing the expression $a[i] > a[j]$ as a constraint. This approach is complex to handle. In our example, it would require that, after every change in the remaining domains for i or j , we would have to check whether there are still possible values for i or j such that $a[i] > a[j]$ can be satisfied. In the worst case that means that we have to check the constraint for all remaining pairs of values for i and j . As a consequence, this approach would be nearly as complex as the first one, the only difference being that the satisfiability check can stop as soon as values for i or j have been detected which satisfy the constraint.

A third approach could delay the check of constraints with symbolic array expressions until the involved indexes assume concrete, constant values. This would be similar to the delayed processing of negation in Prolog [1]. However, in contrast to Prolog, the ongoing computation would still continue. At the latest, the constraint needs to be checked when leaving the encapsulated search space, possibly after implicit labeling. Alternatively, the Muli application could explicitly demand checking delayed constraints, and the MLVM could throw an exception indicating that there are still delayed constraints when trying to leave an encapsulated search before a check. This approach is relatively easy to integrate into the MLVM. However, a major disadvantage of this approach is that time is wasted for exploring parts of the search space which could have been excluded if we had checked the constraint earlier (and found it to be unsatisfiable). Even worse, the corresponding computations could have caused external side-effects which should have never happened. This is a problem since external

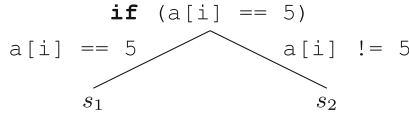


Fig. 1. Excerpt from a search tree, showing branch constraints that involve a symbolic expression for array element access, namely, $a[i]$.

side-effects cannot be reverted on backtracking (e.g., file accesses or console output). Hence, they are discouraged in encapsulated search regions, especially in the case of delayed constraints. Moreover, there is no guarantee that checking the constraint at the end is easier than checking it immediately: If no additional constraints over i and j are encountered in the further evaluation, i and j may still assume the same values. Therefore, the delayed evaluation of the initial constraint is just as complicated as a strict evaluation.

A fourth and last approach could entirely forbid constraint expressions that involve unbound variables as array indexes. However, we feel that this approach is too restrictive. Moreover, it would not really provide new possibilities in Muli.

Unfortunately, all approaches that we could think of have some disadvantages. After comparing the advantages and disadvantages, we implemented the second approach in which constraints are evaluated eagerly for our prototype.

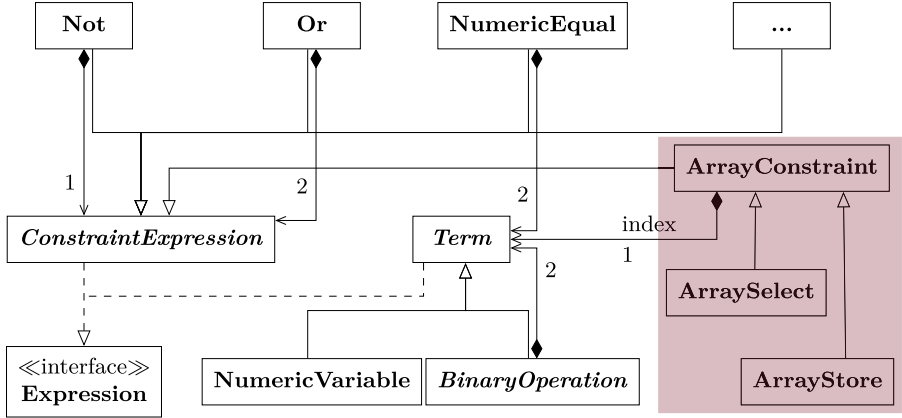
5 Implementation

Implementing the above considerations affects two areas of the runtime environment: First, the solver component must be capable of dealing with constraints over free arrays, i.e., it must be able to check a constraint system that comprises such constraints for consistency as well as to find values for the involved variables. Second, the execution core requires a modified execution semantics of array-related bytecode instructions. Subsequently, we outline our concept for an implementation in the MLVM.

5.1 Modelling Constraints over Free Arrays

Accessing an array element using a free variable as an index, e.g. $a[i]$ with i **free**, yields a symbolic array expression (as described in Sect. 5.2). Using that as part of a condition, e.g., **if** ($a[i] == 5$) { s_1 } **else** { s_2 } causes the runtime environment to branch, thus creating a choice with two branches and appropriate constraints as illustrated in Fig. 1.

The way that a constraint involving symbolic array expressions (such as $a[i] > a[j]$ from Listing 1) can be handled depends on the constraint solver. From the included constraint solvers, only Z3 provides native support for array theories (cf. [15, 16]). Thus, the MLVM had to be extended in order to support such constraints. For handling constraints, the MLVM implements a *solver component* that abstracts from the actual underlying solver. This is achieved by offering a unified interface for the definition of symbolic expressions and constraints.



(Adapted and extended from [2])

Fig. 2. Augmenting the unified interface for the definition of constraints and expressions in order to add symbolic array expressions (additions shaded in red). (Color figure online)

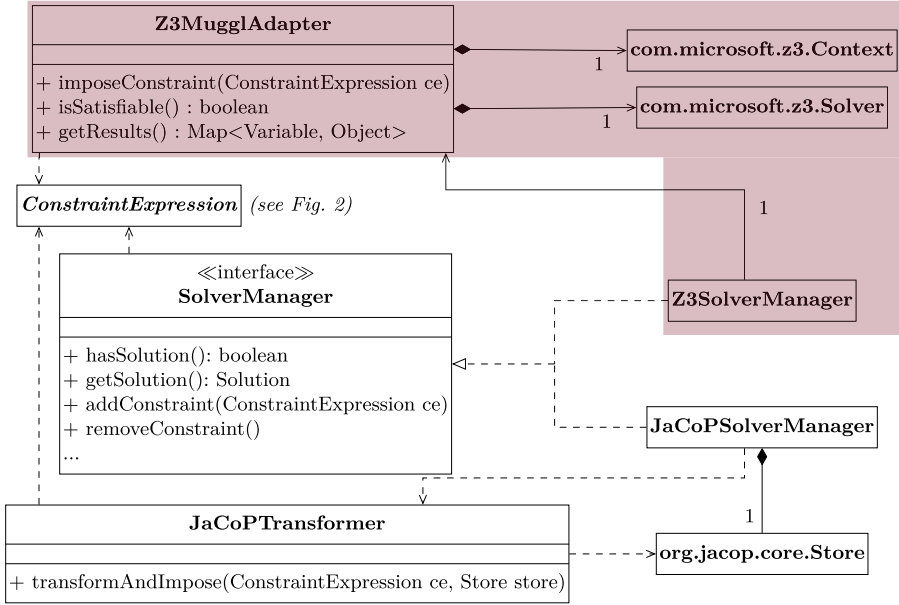
Using a set of transformation methods, the defined constraints are transformed into a suitable representation for the respective solver that can then be queried from the MLVM using an adapter-pattern implementation. As a consequence, it is possible to add support for symbolic array expressions to the unified interface as illustrated in Fig. 2.

The required modifications to the Z3 solver component are illustrated in Fig. 3. In particular, we have added corresponding adapter classes.

The Context type is the main class provided by the official Z3 Java bindings [18]. In order to use it from the MLVM, the `Z3SolverManager` type manages the interaction of the Muli runtime with the Z3 solver by implementing the interface that is expected from an MLVM solver manager and delegating calls to an instance of `Z3MugglAdapter`. The `Z3MugglAdapter` transforms expressions and constraints specified in the unified interface to a corresponding representation for the Z3 Java bindings. For this, the `Z3MugglAdapter` configures and utilizes a Z3 context instance. First, the Z3 context is used to generate a solver. Thereafter, the `Z3MugglAdapter` transforms `ConstraintExpressions` into Z3 objects using the Z3 context instance. For instance, the `Z3MugglAdapter` would transform a symbolic array expression of the form `a[i] == y` (where `a` is a free integer array and `l`, `i`, `v`, `y` are free integers, with `l` being the free length of `a`) into the following (slightly simplified) commands for the Z3 solver²:

```
(declare-const a (Array Int Int))
(declare-const l)
(declare-const i Int)
```

² The semantics of the *select* and *store* array constraints (also see Fig. 2) are further explained in Section 5.3.



(Adapter, extended and simplified from [2])

Fig. 3. Modifications to the solver component of the MLVM in order to integrate the Z3 solver (additions shaded in pink).

```

(declare-const v Int)
(declare-const y Int)
(assert (< i 1))
(assert (>= i 0))
(assert (= v (select a i)))
(assert (= y v))
  
```

For each `ConstraintExpression` of Muli, the corresponding `BoolExpr` of Z3 is added to the Z3 solver's constraint stack. By calling push- and pop-methods of the solver object the Z3 solver can be used in an incremental fashion pushing backtracking points and popping them correspondingly [18].

The Z3 solver has been successfully used in the context of glass-box test case generation, e.g. with the Pex tool [17], even for applications that use symbolic arrays and indexes.

5.2 Conceptual Modifications to Bytecode Execution Semantics

The MLVM executes Java bytecode. Implementing the above considerations requires modifications to the execution semantics of the following bytecode instructions: `Newarray`, `Arraylength`, `Xaload`, `Xastore` (where `X` is replaced with a type, e.g., `Iastore` to store an array element of type `int`) [13].

Newarray is typically used in order to create an array on the heap. For the case of a free array, this requires the creation of an internal representation of the free array, comprising a `NumericVariable` for the length attribute (so that the length of a free array can become part of symbolic expressions) as well as a `Map<Term, T>` that will hold the individual elements. Storing the elements in an `ArrayList<T>` would account for the flexible size of free arrays, yet would conceptually fail for free indexes, since a concrete value for the index is not apparent. This representation will only be used internally by the MLVM. For Muli applications that use a free array its type is equivalent to that of a corresponding regular array.

The `Arraylength` bytecode instruction returns the length of an array [13, § 6.5]. If it is executed in the context of a free array, the instruction has to yield the symbolic representation of the free array's length. As an exception to that, if the logic variable for the length is already bound to a single value, `Arraylength` can return a constant.

The modifications to the `Xaload` and `Xastore` instructions work identically regardless of their type `x` and result in (potentially) non-deterministic execution. The `Xaload` instruction is the bytecode equivalent of accessing a single array element, e.g., `a[i]`, whereas `Xastore` is the equivalent of assigning a value to an array element, e.g., `a[i] = x`. Execution requires to make a distinction based on what is known about the length n of the involved free array (e.g., from constraints that have already been imposed on n). For `a[i]`, if `i` is definitely within the range $(0..n - 1)$, the behaviour is deterministic and results in a `ArraySelect` or `ArrayStore` constraint accordingly. Similarly, if `i` is outside of that range, the execution (deterministically) results in throwing a runtime exception of the type `ArrayIndexOutOfBoundsException`. In all other cases, the execution results in the creation of a non-deterministic choice, distinguishing successful access (yielding a symbolic expression) and the error case (yielding an exception) as alternative outcomes. Each alternative results in imposing appropriate constraints over `i` and n . Using backtracking, the MLVM will evaluate both alternatives successively.

5.3 Prototypical Implementation of Bytecode Execution Semantics

We have implemented a prototype which enables the execution of Muli programs with free arrays using the Z3 solver [15, 16]. The implementation of free arrays extends the `Arrayref` class used by Muli to represent arrays. The resulting `FreeArrayref` class is used by the adjusted bytecode to distinguish between deterministic array behavior (`Arrayref`) and non-deterministic array behavior (`FreeArrayref`), i.e., allowing a variable sized array which accounts for free indexes. When declaring a free array, such as `int[] idx free`, a `FreeArrayref` is automatically generated, carrying a `Map<Term, Object>` for the elements in the array and a `NumericVariable` to represent its length. Terms here represent either an `IntConstant` for a concrete given value, or a `NumericVariable` for free variables.

The bytecode instruction `Arraylength` returns the length of the `FreeArrayref` as a possibly unbounded integer variable. In contrast, for usual `Arrayrefs` an integer constant is returned. More interestingly, the bytecode of `Xaload` and `Xastore` instructions has been adjusted to work with free arrays.

The procedure for all `Xaload` instructions has been implemented uniformly and is divided into two steps. In a first step two constraints are composed and evaluated. The first constraint represents a valid index access, i.e., the index must be larger or equal to zero and smaller than the length of the array. Furthermore, a *select* constraint (`= v (select a i)`) (see Sect. 5.1) is created. For this, first, if the current index is not present in the current free array a new free variable is generated at the index position of the array. If, for example, the free array contains integer values, a new free integer variable is generated. The result is stored in the `Map<Term, Object>` map using the `Term`, either an `IntConstant` or a `NumericVariable`, which is given as the index. On the other hand, if the index already resides within the free array, the value for it is used in the following. These values, i.e., the array, the index, and the value are then represented by objects of the Z3 solver: The free array is mapped to an array value `a`, the index to a value `i`, and the loaded value to a value `v`. In the further course of the program, each of the program's variables is represented by a Z3 counterpart, i.e., new *select* constraints will also target the same Z3 objects `a`, `i`, and `v`. The *select* constraint enforces that at index `i`, where `i` is not necessarily bound to a value yet, value `v` is positioned. This *select* constraint is then conjoined with the aforementioned index constraint.

The second constraint created in `Xaload` instructions represents constraints for an invalid index access, i.e., an access which would raise an `ArrayIndexOutOfBoundsException`. For this, the index is either larger or equal to the length of the free array or the index is less than zero. The satisfiability of both constraints is evaluated. The result of these satisfiability checks is a potential branching point for the program: If either check holds, a choice point of one or two choices is generated. The implementation of the `Xaload` instructions will then proceed with its second step: Each of the feasible branches receives a corresponding marker. The marker indicates which of the cases is to be executed, i.e., whether an exception is to be thrown or if the usual semantics of the `Xaload` instruction [13] should be executed with the constraints given for the respective branch.

`xastore` behaves similarly and its execution is also divided into the two corresponding steps. Aside from its semantics, `xastore` instructions differ from `Xaload` instructions in that the former do not push a *select* constraint for valid index accesses. Rather, they enforce a *store* constraint: Expressions similar to `(= (store a1 i v) a2)` are pushed to the constraint solver. This constraint states that a value `v` is inserted into an array `a1` at index `i`. In this context, `a1` is immutable. Hence the output is a new array `a2` which differs from `a1` in the inserted value. As a consequence, we utilize `a2` as the new representation for the respective free array. For this branch of execution, future constraints are connected with `a2`. If the `xastore` instruction is backtracked, we again utilize

a1 as the representation of the free array. By doing so, we enable the treatment of mutable arrays using the Z3 solver.

Checking constraints is currently done eagerly, i.e., our prototype implements the second approach mentioned in Sect. 4. Lastly, we enabled to skip the creation of `ArrayIndexOutOfBoundsException` to allow for more concise formulations. Take, for example, Listing 5: The domain of values from the array `idx` is not limited. In consequence, each access `usedIdx[idx[i]]` potentially is illegal and we would have to restrict the domain of each value stored in `idx`. By setting a corresponding flag, the program will only regard valid index accesses.

5.4 Demonstration of Prototypical Implementation

We validated our implementation by means of several JUnit tests among which there is the `SimpleSort` example from Listing 5. `SimpleSort` will be discussed in the following. It is deemed a sound proof-of-concept example since it utilizes a manifold of features of free arrays while being intuitive to understand. These used features are comprised of indirect free index accesses using an index array, store and select constraints with free indexes (all within lines 8–10), as well as select constraints with concrete indexes (line 12). However, the example obviously is rather an illustrative choice than a competitive sorting algorithm, since it simply tries out multiple permutations of the input array. As such, the computation time is expected to vastly increase with the input list’s size. We executed the `SimpleSort::sort` algorithm with a list size of 10, 20, 40, and 80 elements, disregarding potential `ArrayIndexOutOfBoundsException`, using an AMD Ryzen 5 4500u CPU running Ubuntu 20.10. Each scenario has been executed five consecutive times. The examples as well as the test suite and code for free arrays in Muli are available under the GPL-3.0 license³ (Table 1).

Table 1. Run times and standard deviations of executing `SimpleSort` with a varying number of elements.

Scenario	Mean run time (s)	Standard deviation of run time (s)
SimpleSort10	2.36	0.02
SimpleSort20	3.41	0.04
SimpleSort40	16.13	0.16
SimpleSort80	283.11	7.43

All in all, the run time behaves as expected. The execution time increases exponentially with the input size. That excludes the step from the `SimpleSort10` to the `SimpleSort20` scenario and might be explained by additional startup overhead including warming up the Java virtual machine. Still, the results produce a valid output.

³ <https://github.com/wwu-pi/muli/tree/free-arrays>.

5.5 Limitations of Prototypical Implementation

Currently, the prototype does not support free arrays of free objects. The reason for this is the inherent complexity and mutability of free objects [6]. Free objects do not always have a deterministic type. Hence, their attributes and attribute types might differ between possible concrete classes. Still, they would have to be represented as the content of an array within Z3. Furthermore, similar to mutable arrays which we did address in this paper, mutable objects might prove difficult to represent in Z3: Each free object which is stored in a free array would have to be represented as a tuple of values. If an object which resides in a free array is to be altered, said representing tuple in Z3 must be altered as well while taking backtracking into account. In the future, we want to design an efficient approach to counteract this issue.

We also plan to implement the third approach of considering array constraints, as discussed in Sect. 4. As a consequence, the Muli runtime is going to allow developers to configure the system in order to choose an approach that best suits their respective search problem. The idea here is to accumulate select and store constraints and only push them to the constraint solver once a concrete solution is requested. Similar to this, comparisons of other approaches, e.g., an own logical layer which enables JaCoP to deal with array constraints, are left for future work. A quantitative evaluation will be able to show whether one approach is generally favourable over the other.

6 Related Work

A first approach to a symbolic treatment of arrays dates back to McCarthy’s *basic theory of arrays* developed in 1962 [14]. It consists of just two axioms, one telling that if a value v is assigned to $a[i]$ then $a[i]$ later on has this value v . The other axiom essentially says that changing $a[i]$ does not affect any other array element. These axioms are clearly not enough for handling free arrays in Muli. McCarthy’s approach was extended to the *combinatorial array logic* by de Moura and Bjørner [16]. It is expressed by a couple of inference rules, which work on a more abstract level and do not address the processing of the search space. Nevertheless, these rules are among the theoretical foundations of Microsoft’s Z3 SMT solver [15]. Based on this solver, support for arrays was included into Microsoft’s test-case generator Pex [17] and into the symbolic code execution mechanism of NASA’s Java Pathfinder, a model checker and test-case generator for Java programs [9]. In order to achieve the latter, Fromherz et al. mainly changed the semantics and treatment of the `xaload` and `xastore` bytecode instructions of their symbolic variant of the Java virtual machine. Their changes to these instructions are similar to our modifications of the MLVM, with the exception that the MLVM has a more sophisticated mechanism for backtracking and resuming an encapsulated search. The authors do not discuss approaches for dealing with the potentially huge search space caused by array constraints.

Also in the context of test-data generation, Korel [11] presented an array-handling approach which avoids the difficulties of free arrays and symbolic array

indexes by resorting to a non-symbolic execution. Korel used a concrete evaluation in combination with dataflow analysis and so-called function minimization in order to reduce the search space. This approach is not suitable for a CLOOP language.

All the mentioned approaches stem from the domains of test-case generation and model checking. To the best of our knowledge, there is no other programming language that offers free arrays with symbolic array indexes.

7 Conclusion and Outlook

As a research-in-progress paper, this work presents approaches for the addition of free arrays to constraint-logic object-oriented programming, thus starting a discussion. The present paper discusses the characteristics and implementation aspects of free arrays. In particular, we address the symbolic treatment of the array length and symbolic array indexes based on constraints. Moreover, we propose a syntax for the declaration and initialization of free arrays. In addition, we discuss ways of dealing with non-deterministic accesses to array elements, proposing possible solutions to that end. The proposed concepts facilitate the use of logic arrays in the context of encapsulated, non-deterministic search that is interleaved with deterministic computations. Muli allows using arbitrary search strategies in order to use symbolic computations that involve arrays.

The foundations for further comparisons and research has been laid by an implementation and demonstration of free arrays for the Muli programming language. To achieve this implementation, the Z3 solver has been added to the MLVM as an alternative backend of the solver components so that symbolic array expressions can be leveraged. This allows for an exhaustive evaluation of the approaches in combination with different solvers in the future.

References

1. Apt, K., Bol, R.: Logic programming and negation: a survey. *J. Log. Program.* **19**(20), 9–71 (1994). [https://doi.org/10.1016/0743-1066\(94\)90024-8](https://doi.org/10.1016/0743-1066(94)90024-8)
2. Dageförde, J.C.: An integrated constraint-logic and object-oriented programming language: the Münster logic-imperative language. Dissertation, University of Münster (2020)
3. Dageförde, J.C., Kuchen, H.: A constraint-logic object-oriented language. In: Proceedings of the 33rd ACM/SIGAPP Symposium on Applied Computing, pp. 1185–1194. ACM (2018). <https://doi.org/10.1145/3167132.3167260>
4. Dageförde, J.C., Kuchen, H.: A compiler and virtual machine for constraint-logic object-oriented programming with Muli. *J. Comput. Lang.* **53**, 63–78 (2019). <https://doi.org/10.1016/j.cola.2019.05.001>
5. Dageförde, J.C., Kuchen, H.: Applications of muli: solving practical problems with constraint-logic object-oriented programming. In: Lopez-Garcia, P., Giacobazzi, R., Gallagher, J. (eds.) *Analysis, Verification and Transformation for Declarative Programming and Intelligent Systems*. LNCS. Springer (2020)

6. Dageförde, J.C., Kuchen, H.: Free objects in constraint-logic object-oriented programming. In: Becker, J., et al. (eds.) Working Papers, European Research Center for Information Systems, vol. 32, Münster (2020)
7. Dageförde, J.C., Teegen, F.: Structured traversal of search trees in constraint-logic object-oriented programming. In: Hofstedt, P., Abreu, S., John, U., Kuchen, H., Seipel, D. (eds.) INAP/WLP/WFLP -2019. LNCS (LNAI), vol. 12057, pp. 199–214. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-46714-2_13
8. Ernsting, M., Majchrzak, T.A., Kuchen, H.: Dynamic solution of linear constraints for test case generation. In: 2012 Sixth International Symposium on Theoretical Aspects of Software Engineering, pp. 271–274 (2012). <https://doi.org/10.1109/TASE.2012.39>
9. Fromherz, A., Luckow, K.S., Păsăreanu, C.S.: Symbolic arrays in symbolic PathFinder. ACM SIGSOFT Softw. Eng. Notes **41**(6), 1–5 (2017). <https://doi.org/10.1145/3011286.3011296>
10. 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>
11. Korel, B.: Automated software test data generation. IEEE Trans. Softw. Eng. **16**(8), 870–879 (1990). <https://doi.org/10.1109/32.57624>
12. Kuchcinski, K.: Constraints-driven scheduling and resource assignment. ACM Trans. Des. Autom. Electron. Syst. **8**(3), 355–383 (2003). <https://doi.org/10.1145/785411.785416>
13. 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>
14. McCarthy, J.: Towards a mathematical science of computation. In: Information Processing, Proceedings of the 2nd IFIP Congress 1962, Munich, Germany, North-Holland, 27 August–1 September 1962, pp. 21–28 (1962)
15. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
16. de Moura, L.M., Bjørner, N.: Generalized, efficient array decision procedures. In: Proceedings of 9th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2009, 15–18 November 2009, Austin, Texas, USA, pp. 45–52. IEEE (2009). <https://doi.org/10.1109/FMCAD.2009.5351142>
17. Tillmann, N., de Halleux, J.: White-box testing of behavioral web service contracts with Pex. In: Bultan, T., Xie, T. (eds.) Proceedings of the 2008 Workshop on Testing, Analysis, and Verification of Web Services and Applications, Held in Conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008), TAV-WEB 2008, Seattle, Washington, USA, 21 July 2008, pp. 47–48. ACM (2008). <https://doi.org/10.1145/1390832.1390840>
18. Z3: Z3 Java Bindings API: Context (2019). https://z3prover.github.io/api/html/classcom_1_1microsoft_1_1z3_1_1_context.html