Structured Traversal of Search Trees in Constraint-Logic Object-Oriented Programming

Jan C. Dageförde[1]([📧]) [iD] and Finn Teegen[2] [iD]

[1] ERCIS, Leonardo-Campus 3, 48149 Münster, Germany
dagefoerde@uni-muenster.de
[2] Institut für Informatik, CAU Kiel, 24098 Kiel, Germany
fte@informatik.uni-kiel.de

Abstract. In this paper, we propose an explicit, non-strict representation of search trees in constraint-logic object-oriented programming. Our search tree representation includes both the non-deterministic and deterministic behaviours of executing an application. Introducing such a representation facilitates the use of various search strategies. In order to demonstrate the applicability of our approach, we incorporate explicit search trees into the virtual machine of the constraint-logic object-oriented programming language Muli. We then exemplarily implement three search algorithms that traverse the search tree on-demand: depth-first search, breadth-first search, and iterative deepening depth-first search. In particular, the last two strategies allow for a complete search, which is novel in constraint-logic object-oriented programming and highlights our main contribution. Finally, we compare the implemented strategies using several benchmarks.

Keywords: Constraint-logic object-oriented programming · Explicit search tree · Complete search strategy · Virtual machine implementation

1 Motivation

In constraint-logic object-oriented programming, combining imperative code with features from logic programming causes the runtime to execute parts of the imperative code non-deterministically ("don't know" non-determinism). To give an example, the program (or search region) depicted in Listing 1 has two solutions. The example is written using the Münster Logic-Imperative Language (Muli), which we explain in Sect. 2. The search region declares a boolean logic variable `coin`. Subsequently, evaluating the `if` statement causes the runtime environment to take and implement a decision regarding the potential value of `coin`, thus introducing non-determinism. Consequently, implementing the decision selects a single branch of execution, eventually resulting in one of the two outcomes.

```
boolean flipTwoCoins() {
    int coin1 free, coin2 free;
    if (coin1 == 0)
        return false;
    else if (coin2 == 0)
        throw Muli.fail();
    else
        return true; }
```

```
boolean flipCoin() {
    int coin free;
    if (coin == 0)
        return false;
    else
        return true; }
```

Listing 1. A simple non-deterministic search region in Muli for the demonstration of constraint-logic object-oriented programming concepts.

Listing 2. Muli search region example that comprises two solutions and a failure.

Non-deterministic execution is useful for applications involving search, i. e., an application would usually cause the runtime environment to evaluate more than one branch. To that end, the runtime environment systematically evaluates multiple alternative branches in sequence. Non-deterministic branching dynamically creates an implicit search tree that represents the various execution paths that lead to alternative outcomes of a program. The goal of the present work is to make this search tree explicit at runtime. It encodes the various execution paths of a program, the choices encountered along every path, and every path's outcome (i. e., solution or failure). As there can be paths of infinite length, our search tree representation is non-strict. Our search tree then serves as a basis for structured traversal by arbitrary search algorithms, including iterative deepening depth-first search. Furthermore, by making the search tree explicit, it is possible to inspect the search tree at any given point in time, e. g., after search or even at an intermediate stage. This way, the search tree aids in effective debugging.

This paper provides the following contributions:

– A general *search tree structure* for constraint-logic object-oriented programming that encapsulates execution state (Sect. 4).
– *Search algorithm implementations* that traverse the search tree structure for finding solutions to constraint-logic object-oriented programs (Sect. 5).
– A discussion of the implications of our work for executing object-oriented (imperative) programs non-deterministically (Sect. 6).

First of all, Sect. 2 introduces concepts of constraint-logic object-oriented programming, followed by an outline of the Muli virtual machine in Sect. 3.

2   Constraint-Logic Object-Oriented Programming

Constraint-logic object-oriented programming combines the flexibility of imperative and object-oriented programming with features from constraint-logic programming, namely logic variables, constraints, and search. Muli is a constraint-logic object-oriented programming language that is based on Java [4].

In Muli, *logic variables* are declared in a way that is similar to declaring regular variables. As indicated in Listing 1,

```
int coin free;
```

declares a logic variable of a primitive (integer) type. Instead of assigning a constant value, the **free** keyword specifies that `coin` is a logic variable. A logic integer variable can be used interchangeably with other integer variables, i.e., they can become part of conditions or arithmetic expressions and can be passed to methods as parameters [3]. In contrast to regular variables, logic variables are used symbolically. Recent work is looking into support for reference-type logic variables [2], but here we focus on logic variables of primitive types.

Constraints are defined as relational expressions, (typically) involving logic variables. For simplicity, Muli does not provide a dedicated language feature for imposing constraints. Instead, a constraint is imposed whenever the flow of execution branches, such as when a branching condition is evaluated. Therefore, constraints are derived from boolean expressions. For instance, in Listing 1

```
if (coin == 0) { s₁ } else { s₂ }
```

`coin` occurs in the condition and is not sufficiently constrained, so that the condition can be evaluated to either true or false. As a result, the evaluation of the condition creates a *choice*, from which alternatives are evaluated non-deterministically. The runtime environment selects an alternative by imposing the corresponding constraint. In our example, by imposing $coin \neq 0$ the runtime environment can proceed with the evaluation of s_2. The runtime environment leverages a constraint solver for finding solutions as well as for cutting execution branches early as soon as their constraint system becomes inconsistent.

Search transparently performs non-deterministic evaluation in combination with backtracking until a solution is found. Implicitly, following a sequence of choices (and taking decisions at each choice) produces a (conceptual) search tree that represents the order of execution. In such a search tree, inner nodes are choices and leaves represent alternative ends of execution paths. In Muli, an execution path ends with a *solution* (specified by either **return** or **throw**) or with a *failure*, e.g., if a path's constraint system is inconsistent. The full listing of our example in Listing 1 demonstrates how solutions are returned. After search completes, solutions of the example are **false** and **true** (in any given order).

Moreover, applications sometimes require an *explicit failure* denoting the end of an execution path without a solution. In Muli, an explicit failure is expressed by **throw** `Muli.fail()`. Nevertheless, executing that statement will not return an exception. Instead, the statement is specifically interpreted by the runtime environment, resulting in backtracking. Listing 2 provides a slightly extended search region with three execution paths, one of which ends in a failure.

The main program is executed deterministically, whereas all non-deterministic search is *encapsulated*. Encapsulation gives application developers control over search. In addition to coarse-grained control (i.e., requesting either a single solution or an array comprising all solutions), Muli offers

fine-grained control by returning a Java stream that evaluates solutions non-strictly. `Muli.`**`muli`**`()` accepts a `Supplier` and returns a stream of `Solution` objects. In Java (and, therefore, in Muli), a `Supplier` denotes either a lambda expression or a method reference (both without arguments). We refer to the method that is passed as an argument as a *search region*, as it will be executed non-deterministically and therefore describes the constraint-logic object-oriented problem. Following the principles of the Java Stream API, solutions can be retrieved from the stream individually on demand [5]. For instance, considering Listing 1, a stream is initialised using

```
Stream<Solution<Boolean>> stream = Muli.muli(self::flipCoin),
```

and the actual search starts as soon as the first solution is requested from the stream.

3 Muli Logic Virtual Machine

The Muli Logic Virtual Machine (MLVM) is a runtime environment for Muli. The MLVM is a custom Java Virtual Machine (JVM) that complies with the JVM Specification (see [10]) for deterministic execution. Moreover, it adds modifications that support Muli-specific extensions, particularly symbolic execution and non-deterministic execution [4]. As in a regular JVM, execution state is represented in the MLVM by a combination of *program counter (PC)*, a *heap*, a stack of executed method frames (*frame stack*), and an *operand stack* per frame. Additional state serves the purpose of supporting non-deterministic execution and constraints. In particular, this includes the constraint stack and the trail.

The *constraint stack* maintains the active constraint system, i.e., the conjunction of all constraints on the stack [4]. Representing the constraint system in a stack structure is beneficial as constraints are added dynamically during execution. Consequently, on backtracking, only the most recently added constraints need to be removed from the stack. Moreover, the *trail* records changes that are made to the virtual machine (VM) state during execution. On backtracking, the information on the trail can be used to revert to a previous execution state. More precisely, using the trail, backtracking achieves the specific state of the choice at which the next decision can be made. In fact, the trail is therefore split up into incremental trails, one per choice, each describing how to backtrack towards the next choice. In addition, in order to be able to not only backtrack to a choice (upwards along a search tree) but to achieve an arbitrary previous state (including downward navigation), the MLVM maintains two trails per choice, one being the inverse of the other [5]. In the following, we call the trail for backtracking *backward trail*, as opposed to the *forward trail* that is used to navigate downwards.

Like a regular JVM, the MLVM reads applications from bytecode and executes bytecode instead of the original source. Muli's bytecode format is compatible with that described in [10], merely adding custom attributes in order to represent logic variables [4]. For instance, the example application from Listing 2

```
 0: iload_1                // coin1
 1: iconst_0
 2: if_icmpne ⑦           // coin1 != 0
 5: iconst_0
 6: ireturn                // return false
 7: iload_2                // coin2
 8: iconst_0
 9: if_icmpne ⑯           // coin2 != 0
12: invokestatic  #91    // fail()
15: athrow
16: iconst_1
17: ireturn                // return true
```

Listing 3. Bytecode generated by the Muli compiler for the program in Listing 2.

Table 1. Bytecode instructions that may cause non-deterministic branching upon execution. <cond> is a placeholder for specific comparisons, e. g., eq for equality.

Triggering bytecode instruction	Type of choice	No. of decisions
`If<cond>, If_icmp<cond>`	**if** instruction, integer comp	2
`FCmpg, FCmpl, DCmpg, DCmpl`	floating point comparison	2
`LCmp`	long comparison	3
`Lookupswitch, Tableswitch`	**switch** instruction	1 per case + 1

compiles to the bytecode instructions in Listing 3. Some bytecode instructions exhibit non-deterministic behaviour. For instance, if_icmpne in Listing 3 jumps to the specified instruction if the two integer operands on the operand stack are not equal. Otherwise, execution continues linearly with the following instruction. If one or both operands are logic variables, both *jumping* and *not jumping* are feasible alternatives. As logic variables are used in the current example, the execution of if_icmpne instructions creates choice points that offer two decision alternatives. While **if** instructions always provide two alternatives (i. e., jumping to the **else** branch or not), **switch** instructions result in alternatives according to the number of **case**s plus one for the **default** case, each jumping to instructions accordingly. Table 1 provides a reference of instructions that may exhibit non-deterministic behaviour and counts the decision alternatives from which the MLVM chooses.

Executing a bytecode instruction with non-deterministic branching creates a choice point in the MLVM [4]. Prior to this work, the implementation of the choice point itself was responsible for managing the execution of its branches. More specifically, executing a bytecode instruction created a choice point representation in the MLVM. Consequently, the created choice point contained information about applicable branches, but also implemented the behaviour of search.

That is, upon creation, the choice point representation immediately selected the first decision alternative and applied it, thus committing to a specific branch. The created choice point representations are stored in a stack of choice points. The MLVM referred to the choice point stack during backtracking. Starting from the top, it popped choice points until reaching one with an alternative that had not been evaluated yet. It then immediately committed to this alternative by adding its constraint and following its path.[1] As a consequence, the runtime environment never actually stored an explicit representation of the search tree. Instead, the choice point stack merely maintained a single path through the (implicit) search tree. Therefore, diverting from the currently executed path was not possible, effectively restricting the search capabilities of the MLVM to depth-first search. All things considered, the previous MLVM used a complex, tangled mixture of responsibilities in which bytecode-instruction implementations, choice-point implementations, and the VM realise non-deterministic search in combination.

In a cleaner architecture,

– declaratively executing a bytecode instruction creates choice objects and just returns them to the MLVM (instead of performing a decision right away), and
– choice objects only hold information about available decision alternatives (but no implementation for taking decisions).

As a consequence, the MLVM is the only element that is allowed to change execution state by committing to decisions, instead of sharing this permission with choice objects or instruction implementations. The search tree structure that we discuss subsequently facilitates an explicit representation that holds a declarative representation of choices and of the alternatives that each choice provides. Overall, the structure serves as a clean basis for following arbitrary execution paths through the tree.
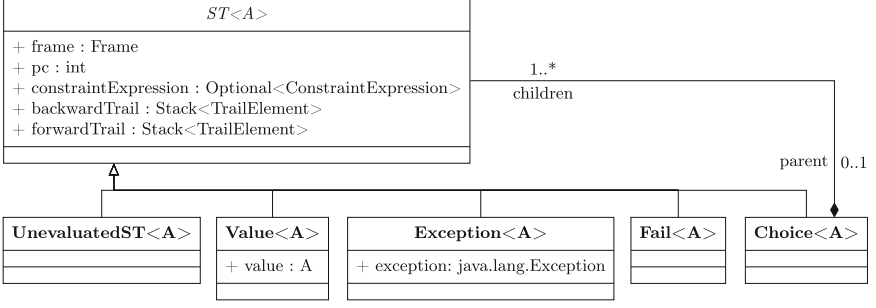
4   Search Trees

A declarative, explicit search tree representation lays the groundwork for following arbitrary execution paths instead of limiting execution to depth-first search only. We first explain the conceptual representation, outlining the intuition of the elements that constitute the search tree. Afterwards, we describe how a search tree is constructed dynamically during the execution of a Muli application. Last, we abstractly describe navigation through the search tree as the basis for search.

4.1   Representation

Conceptually, our explicit search tree comprises five distinct node types. There are node types for returned values, thrown exceptions, choices between non-deterministic branches, failed computations, and yet unevaluated search trees.

---

[1] Provided that the constraint system was still consistent. Otherwise, backtracking occurred until the next choice point that offered an unevaluated, feasible alternative.

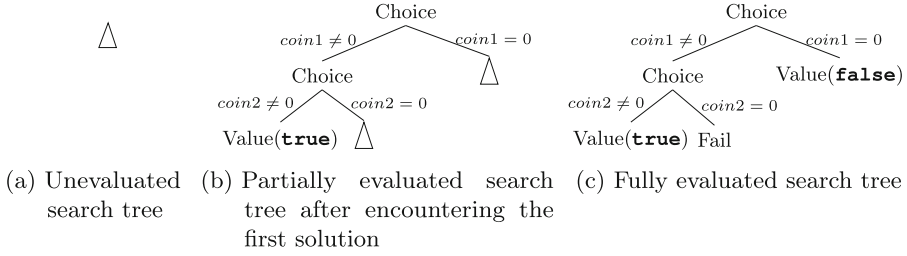**Fig. 1.** Class diagram for the representation of search trees.

Figure 1 shows a class diagram for our search tree representation. Basically, this representation corresponds to an algebraic data type and therefore does not implement any decision-taking in contrast to the previously used choice points.

As solutions of a search region, a `Value` node holds the value returned by a computation while an `Exception` node does the same with an exception that has been thrown. A `Fail` node represents either an explicit failure or branches whose constraint system is inconsistent. As a consequence, it does not hold any values. Furthermore, `Choice` nodes store a list of subtrees which, in turn, reference their `parent` choice. Having an explicit reference to each node's parent allows for easy and direct navigation through the search tree. For the root node of a search tree, the `parent` attribute is **null**. Finally, `UnevaluatedST` serves as a proxy for subtrees that have not been evaluated yet, facilitating non-strict usage.

Moreover, each node in the search tree stores fields that prepare for later execution. The `frame` and `pc` fields represent a reference to the (mutable) stack frame and the value of the PC at which the node has been created. Each node holds an optional constraint expression that has to be satisfied in order to reach this node, e.g., as a consequence of non-deterministic branching. Additionally, the backward trail stores the changes to the VM state that were made in order to reach this node (thus preparing for backtracking), whereas the forward trail stores changes that are needed in order to return to this node afterwards. In combination, these fields are used to properly manipulate the state of the MLVM during the traversal of the search tree, which is discussed in detail in Sect. 4.3.

4.2   Construction

The actual search tree is constructed during search. A search strategy is responsible for determining the order in which the search tree is traversed. Regardless of the order, a search strategy evaluates `UnevaluatedST` nodes as long as there are such nodes left and the encapsulating program demands additional solutions. In general, the MLVM evaluates an `UnevaluatedST` node by imposing the node's constraint and executing the bytecode of the search region starting from the PC, which the node points to, until either of the following situations occurs.

(a) Unevaluated search tree   (b) Partially evaluated search tree after encountering the first solution   (c) Fully evaluated search tree

Fig. 2. Different evaluation stages of the search tree corresponding to the search region in Listing 2. The constraint of each subtree is noted at the respective edge.

- The computation in the search region returns with a value,
- an uncaught exception occurs during execution,
- the method `Muli.fail()` signals a failed computation, or
- one of the instructions in Table 1 is executed, which results in the creation of a `Choice` object.

In any case, the `UnevaluatedST` node in the search tree is replaced by its evaluated counterpart, i. e., by a `Value`, `Exception`, `Fail`, or `Choice` node. Note that all children of a newly created `Choice` node are unevaluated search trees initially. Furthermore, state changes that were made during this evaluation are received from the MLVM and stored within the new node as its backward trail.

At the beginning of search, the search tree is unknown and therefore initially represented by a single `UnevaluatedST` node. The PC of that node points to the start of the search region, and the optional constraint expression is left empty since no constraints apply to the start of a search region. Similarly, the trails are empty as this node has not yet been evaluated. Figure 2 exemplarily shows three search trees for the program from Listing 2 that all are evaluated to a different degree, and thus illustrate various intermediate evaluation stages that can occur during a search. The illustration assumes a depth-first search strategy; therefore, other search strategies will result in different intermediate stages.

4.3 Traversal

The implementation of any search algorithm requires to be able to navigate through the search tree in any direction, i. e., upwards and downwards. For example, if a branch of a search tree has been fully evaluated, search continues elsewhere. While navigating through the search tree, it is vital to ensure that the MLVM remains in a consistent state. A node's forward and backward trail together with its frame and PC are used for that purpose. In general, navigation takes place from an already evaluated node to another evaluated node, since only evaluated nodes have a trail (see Sect. 4.2). More specifically, a `Choice` node is always the target node or source node when navigating upwards or downwards.

We navigate upwards in a search tree by following references to the parents until we reach the target node (e. g., the root), backtracking the VM state in

```
void navigateUpwards(ST from, Choice to) {
  while (from != to) {
    if (from.constraintExpression.isPresent())
      constraintStack.pop();
    vm.processTrail(from.backwardTrail, from.forwardTrail);
    vm.setFrame(from.frame); vm.setPc(from.pc);
    from = from.parent; } }

void navigateDownwards(Choice from, ST to) {
  Stack<ST> nodes = new Stack<>();
  while (to != from)
    nodes.put(to); to = to.parent;
  while (!nodes.empty()) {
    to = nodes.pop();
    vm.setFrame(to.frame); vm.setPc(to.pc);
    vm.processTrail(to.forwardTrail, to.backwardTrail);
    if (to.constraintExpression.isPresent())
      constraintStack.push(to.constraintExpression.get()); } }
```

Listing 4. Methods for navigating upwards and downwards in a search tree.

the process. In doing so, we remove previously imposed constraints from the constraint stack and undo the changes to VM state by processing the backward trails of nodes along the path. At the same time, the backward trails are converted into forward trails so that a node from which we navigate away can be reached again later when navigating downwards, e.g., for the evaluation of another subtree of that node. Last but not least, the frame and PC of the VM are set accordingly, using the information that was recorded at each node when it was created.

Navigating downwards is slightly more complicated as we first need to determine how to reach a target node from the current (source) node. However, we always have a reference to the target. Therefore, we can use the target's parents in order to find the path to the source. Afterwards, we process the path in reverse order, thus getting from the source node to the target node. We basically do the opposite of what is done in upwards navigation: For each node, we set the frame and PC to what is recorded in the node, apply the forward trail to reapply changes to the execution state, and impose a node's constraint if present. Simultaneously to processing the forward trail, we convert it again into a backward trail to be later able to navigate upwards. For clarity, Listing 4 shows simplified implementations for navigating upwards and downwards, respectively. Subsequently, these general navigation methods serve as primitives for traversal.

```
Choice findCommonAncestor(ST a, ST b) {
  initialise empty set;
  while (b != null) {
    add b to set;
    b = b.parent; }
  while (!set.contains(a))
    a = a.parent;
  return a; }
```

Listing 5. Algorithm for finding the first common ancestor of two nodes.

5   Search Strategies

As a demonstration of how the explicit search tree representation can be employed for the implementation of search strategies, we outline the implementations of three particular ones.

Depth-First Search. The implementation of depth-first search maintains a stack of unevaluated subtrees from the search tree. At the beginning of the search, the initial node (see Sect. 4.2) is pushed to the stack. Then, depth-first search repeatedly pops an unevaluated search tree node from the stack and tries to evaluate it. If its evaluation results in a `Choice` node, its children are pushed to the stack and search continues by popping the next node from the stack (i.e., a local subtree). Otherwise, if a `Value` or `Exception` node is encountered, the search strategy must be able to return the result to the encapsulating program. To that end, it reverts execution state to the state from the beginning of search using `navigateUpwards`. When search is picked up again, the search strategy uses `navigateDownwards` in order to evaluate the next node from the stack. Finally, if the node at hand is evaluated to a `Fail` node, local backtracking is performed, i.e., we navigate upwards to the nearest parent that has at least one unevaluated subtree.

Breadth-First Search. Instead of a stack, a FIFO queue keeps track of unevaluated subtrees. Beginning or resuming search dequeues nodes from the head of the queue. In contrast, when a `Choice` node is encountered, its children are enqueued at the end. Another difference is the fact that breadth-first search requires navigating between arbitrary nodes within the search tree. While it is, of course, possible to go over the root node, it is more efficient to navigate along a path going over the first common ancestor of the two involved nodes. Listing 5 shows a simple algorithm that determines the first common ancestor of two nodes in the search tree. Once the first common ancestor is found, search combines `navigateUpwards` (to the found ancestor) and `navigateDownwards` in order to efficiently navigate between two arbitrary nodes.

Iterative Deepening Depth-First Search. Our search tree can also be used to implement an exciting variant of iterative deepening search. Iterative deepening provides the strength of depth-first search while ensuring that solutions can be found even if there are execution paths of infinite length. In iterative deepening, search is bounded by a constant maximum depth. Search proceeds in a depth-first manner until nodes are reached that are at the maximum depth. In that case, search first evaluates other nodes up to that depth, thus assuming breadth-first search behaviour. Only if additional solutions are required, search increases the bound, again by a constant, and so on. In Muli, aided by the inverse trails, when the bound is increased, the runtime environment does not need to restart computation at the root, which usually leads to a reevaluation of known execution paths (and solutions). Instead, it leverages the (partial) search tree and the recorded inverse trails in order to restart computation from known states that provide further alternatives.

6   Discussion

The implementation of our search tree structure in the MLVM facilitates the non-deterministic execution of imperative (object-oriented) programs in novel ways, using search strategies that could not be implemented without an explicit structure. The existing depth-first search strategy has been reimplemented and is now based on the explicit search tree structure as well. In order to ensure that the required changes do not adversely affect the performance of depth-first search, we first compare the runtime behaviour before discussing novel aspects of search. Note that we measure only performance, not memory consumption. Obviously, maintaining the search tree requires more memory than merely storing the current execution path. However, a possible memory optimisation would be to discard search tree nodes that belong to exhaustively evaluated subtrees—especially in depth-first search strategies.

We are interested in comparing the performance of depth-first search in the new search-tree-based and old choice-point-stack-based implementations. To that end, a set of experiments is conducted in a modified MLVM that contains our search-tree structure as well as in an MLVM without modifications, each executed by OpenJDK 1.8.0_212.[2] Since the MLVM is executed by a JVM, we drop the first 15 executions in order to account for effects caused by just-in-time compilation and take the performance values of subsequent executions. In total, we aggregate performance values of 500 executions per experiment, tackling classic search problems. The first experiment calculates a solution to the 3-partition problem for a fixed set of integer values using a depth-first search strategy. Until the first solution is found, search passes 374 choices. The second finds a solution to the Send More Money puzzle. For reference, we also execute corresponding Curry implementations on PAKCS 2.1.1 using depth-first search. Figure 3 features the average execution times. Our experiment indicates that the

---

[2] Ubuntu 18.04.2 with 4.15.0 x86_64 GNU/Linux kernel; Intel Core i5-5200U CPU.
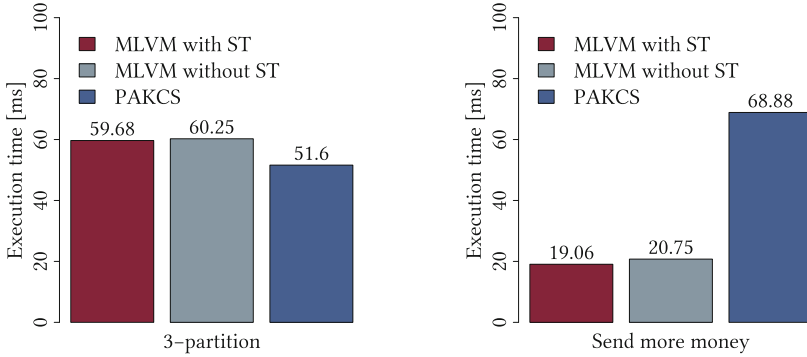
Fig. 3. Comparison of execution times in MLVM with or without explicit search trees, both using depth-first search. Execution times in PAKCS for reference.

```
private static boolean nonTerminatingCoin() {
  int coin free;
  if (coin == 0)
    return true;
  else
    return nonTerminatingCoin(); }
```

Listing 6. Muli search region featuring an infinite amount of execution paths.

implementation and use of an explicit search tree do not negatively affect depth-first search performance. Moreover, the comparison to PAKCS is encouraging, seeing that Muli search regions offer competitive performance while providing support for using side-effects during non-deterministic execution.

Since the use of explicit search trees does not add visible overhead to execution times, we can focus on the benefits of using a search tree representation at runtime. The MLVM now features additional search algorithms beyond depth-first search that all leverage the search tree structure. In particular, using breadth-first search is novel to the non-deterministic execution of imperative programs that have side-effects.

Consider the search region from Listing 6. For lack of a termination condition, there is one infinite execution path. Therefore, it is impossible to evaluate the search tree (or the application) strictly. In our depth-first search implementation, the infinite execution path is the leftmost one. As a result of this structure, depth-first search is unable to compute a single solution. In contrast, several solutions can be returned using a breadth-first or iterative deepening strategy, even though the tree can never be evaluated in full. As a more sophisticated example, we have implemented a search region that finds solutions to the Water jugs problem. Here the MLVM is unable to evaluate a full search tree as there

Table 2. Comparison of search strategies w. r. t. the number of solutions that are returned within ten seconds.

	DFS	BFS	ID-DFS
Simple infinite recursion	0	1469.7	1555.2
Water jug problem	0	29.5	34.4

are cyclic execution paths that result in valid solutions or failures. We have executed these programs using the available strategies 500 times for up to ten seconds each and indicate the average number of solutions in Table 2.

Note that the results do not imply that depth-first search is generally a bad strategy. On the contrary, the combination of increased memory requirements and the time needed for changing VM state using the trail still speaks against using breadth-first search by default. Iterative deepening shares this disadvantage in case that additional levels of the search tree need to be evaluated (but is as efficient as depth-first search if the initial depth is sufficient). Consequently, the results indicate that iterative deepening depth-first search is a good trade-off, if not a better strategy. Further evidence is needed to conclusively argue that iterative deepening is a superior strategy in general. In any case, both are useful strategies in certain situations in which depth-first search falls short.

The search tree structure that is presented in this paper is conceptually similar to the ST structure known from the KiCS2 compiler for Curry [7]. However, Curry search trees only encode evaluation alternatives of an expression. In contrast, search trees for constraint-logic object-oriented programming need to encode the execution behaviour, i. e. VM state changes, that results from different alternatives. Consequently, the state changes are recorded on the corresponding paths that lead to solutions, so that the VM can change state depending on the alternative that is being evaluated. In our current work, we do this by maintaining the forward and backward trails on edges of the search tree.

Prior to our work, the execution state of constraint-logic object-oriented programming in Muli was represented by the PC, frame stack, operand stacks, constraint stack, trail, and choice point stack. Our work results in a slightly altered definition of execution state. What previously was a choice point stack is now replaced by the search tree and a pointer to the current search tree node that is under evaluation. In addition, a search algorithm is responsible for maintaining a suitable data structure that keeps track of the progress of traversing the search tree, e. g., a stack of not-yet-evaluated choices in depth-first search algorithms.

Moreover, the explicit search tree structure is useful for the development of constraint-logic object-oriented programs, as it can be helpful to visualise the structure of search. Specifically, we can visualise at which points different kinds of choices are introduced and which solutions are encountered by the runtime environment. During the development of the MLVM, the search tree structure is useful for ensuring that non-deterministic branching and search algorithms are implemented correctly. In contrast, the structure of the previous approach

impeded the diagnosis of problems with non-deterministic execution, as only the current execution path was represented. Consequently, relevant information about previously encountered choices and solutions was lost, whereas this information is adequately represented in the explicit search tree. All in all, the discussed benefits of an explicit search tree structure outweigh the increased memory requirements.

7   Related Work

For software testing, symbolic execution trees describe possible execution paths of an imperative program under test [8,11]. Similar to our search tree, a symbolic execution tree represents choice points where execution branches and collects path constraints. However, a symbolic execution tree usually describes the entire execution of an application. In contrast, our search tree for constraint-logic object-oriented programming describes the execution of specific application parts, namely the non-deterministic execution of a search region. Its leaf node types are tailored to describing the result (i. e., solutions or failures) of execution paths. Moreover, a symbolic execution tree is the result of performing depth-first search, whereas the dual trails of our search tree specifically support arbitrary traversal.

The idea of using an explicit data structure for non-deterministic computations in order to facilitate different search strategies is extensively used in functional logic programming [1,7]. In functional logic programming, search trees cover non-determinism of expressions, i. e., they encode alternatives for the values to which a pure expression can be evaluated. In contrast to that, constraint-logic object-oriented programming is non-deterministic in its execution behaviour, which includes side-effects incurring during execution. Therefore, the present search tree structure has to encode alternative behaviour, including side-effects, in addition to final results. In addition to the representation usually used in functional logic programming, our representation includes node types for exceptions (as a different kind of solution) and unevaluated search trees. The latter types are a prerequisite for the on-demand construction of the search tree during search, which is innately given with the non-strict evaluation in functional logic programming.

An explicit data structure for representing a search tree structure has also been used in a monadic definition of constraint programming [13]. In contrast to our work, it abstracts from side effects and asserts an ordering of subtrees. Another explicit search tree is used for implementing a domain-specific language (DSL) for probabilistic programming in OCaml [9]. As OCaml is strict, the on-demand characteristic of the search tree is modelled explicitly using lambda functions. Although OCaml is not purely functional, the authors disregard backtracking w. r. t. behaviour, modelling only non-deterministic results of pure expressions.

As an alternative to using an explicit search tree, the interface of the probabilistic DSL in OCaml has also been implemented by using continuation passing

style and by using delimited continuations, i. e., using `shift` and `reset` [6]. Using continuations provides an implementation in a direct style and removes the runtime overhead of the search tree data structure. Therefore, implementing Muli by means of `shift` and `reset` is an interesting option for future work. In this case, however, monadic reflection (i. e., inspecting the search tree) is expensive, and its efficient implementation requires additional techniques [12].

The concept of trails has initially been adapted from the trail described for the Warren Abstract Machine (WAM) [14] and has been extended towards dual trails for arbitrary execution state in [5]. Dual trails facilitate their use for backtracking upwards along a search tree as well as for descending towards nodes that have been (partially) evaluated. For their duality, the two trails were originally termed trail and inverse trail. Here we call them backward trail and forward trail, respectively, in order to improve clarity regarding the direction in which they are used. Extending previous work, the present paper leverages dual trails for the implementations of search strategies other than depth-first search.

8   Conclusion and Future Work

Our search tree structure represents the paths of non-deterministic execution of a search region. A runtime environment of a constraint-logic object-oriented language can construct the search tree non-strictly while executing a search region, thus encoding the solutions that are found as well as the execution behaviour of imperative code that leads to solutions or intermediate choices. As a result, the explicit search tree representation can serve several purposes. First, it provides a structure that arbitrary search strategies utilise for traversing the search tree. Furthermore, we found it to make debugging of non-deterministic execution behaviour more productive by allowing developers who use a debugger to introspect intermediate state at breakpoints. More opportunities for utilising the search tree in constraint-logic object-oriented programming will be part of future work.

We also extend Muli's runtime environment, the MLVM, to implement depth-first search, breadth-first search, and iterative deepening depth-first search. Even though they are well-known as search algorithms for tree traversal, they are of particular interest in the context of constraint-logic object-oriented programming where the search tree is not (entirely) known before the program that it represents has been executed in its entirety. The MLVM already supported depth-first search using the previous, unstructured approach, but our evaluation demonstrates that using a structured approach does not add any overhead. On the contrary, the explicit representation provides opportunities for novel search algorithms that could not be used for executing constraint-logic object-oriented programs prior to our work. The modifications have already been integrated into the open source MLVM and are available at https://github.com/wwu-pi/muli.

The current work is the basis for future endeavours. The search tree structure could be used for implementing an interactive search strategy in which a developer could manually decide how to explore the search space when a choice is

encountered. This interactivity could be an additional aid for debugging. Moreover, it is interesting to explore alternatives to explicit search trees, such as the use of delimited continuations for the implementation of non-deterministic execution.

References

1. Braßel, B., Hanus, M., Huch, F.: Encapsulating non-determinism in functional logic computations. J. Funct. Log. Program. **2004**(6) (2004)
2. Dageförde, J.C.: Reference type logic variables in constraint-logic object-oriented programming. In: Silva, J. (ed.) WFLP 2018. LNCS, vol. 11285, pp. 131–144. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-16202-3_8
3. Dageförde, J.C., Kuchen, H.: An operational semantics for constraint-logic imperative programming. In: Seipel, D., Hanus, M., Abreu, S. (eds.) WFLP/WLP/INAP -2017. LNCS (LNAI), vol. 10997, pp. 64–80. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-00801-7_5
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.: Retrieval of individual solutions from encapsulated search with a potentially infinite search space. In: Proceedings of 34th SAC, pp. 1552–1561. Limassol, Cyprus (2019). https://doi.org/10.1145/3297280.3298912
6. Danvy, O., Filinski, A.: Abstracting control. In: Proceedings of the 1990 ACM Conference on LISP and Functional Programming - LFP 1990, pp. 151–160. ACM Press (1990)
7. Hanus, M., Peemöller, B., Reck, F.: Search strategies for functional logic programming. In: Proceedings of ATPS 2012, pp. 61–74, GI LNI 199 (2012)
8. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976). https://doi.org/10.1145/360248.360252
9. Kiselyov, O., Shan, C.: Embedded probabilistic programming. In: Taha, W.M. (ed.) DSL 2009. LNCS, vol. 5658, pp. 360–384. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03034-5_17
10. 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
11. Majchrzak, T.A., Kuchen, H.: Automated test case generation based on coverage analysis. In: TASE 2009. IEEE (2009). https://doi.org/10.1109/TASE.2009.33
12. van der Ploeg, A., Kiselyov, O.: Reflection without remorse: revealing a hidden sequence to speed up monadic reflection. In: ACM SIGPLAN Notices, vol. 49, no. 12, pp. 133–144 (2015)
13. Schrijvers, T., Stuckey, P., Wadler, P.: Monadic constraint programming. JFP **19**(6), 663–697 (2009). https://doi.org/10.1017/s0956796809990086
14. Warren, D.H.D.: An abstract prolog instruction set, Technical repot, SRI International, Menlo Park (1983)