Memoized Pull-Tabbing for Functional Logic Programming

Michael Hanus[0000−0002−4953−8202] Finn Teegen[0000−0002−7905−3804]

Institut für Informatik, CAU Kiel, 24098 Kiel, Germany
{mh,fte}@informatik.uni-kiel.de

Abstract. Pull-tabbing is an evaluation technique for functional logic programs which computes all non-deterministic results in a single graph structure. Pull-tab steps are local graph transformations to move non-deterministic choices towards the root of an expression. Pull-tabbing is independent of a search strategy so that different strategies (depth-first, breadth-first, parallel) can be used to extract the results of a computation. It has been used to compile functional logic languages into imperative or purely functional target languages. Pull-tab steps might duplicate choices in case of shared subexpressions. This could result in a dramatic increase of execution time compared to a backtracking implementation. In this paper we propose a refinement which avoids this efficiency problem while keeping all the good properties of pull-tabbing. We evaluate a first implementation of this improved technique in the Julia programming language.

1 Introduction

Functional logic languages [7] combine the main features of functional and logic languages in a single programming model. In particular, demand-driven evaluation of expressions is amalgamated with non-deterministic search for values. This is the basis of optimal evaluation strategies [4] and yields a tight integration between specifications and code [8]. However, it also demands for advanced implementation techniques—an active research area in declarative programming. This paper proposes a new implementation model which combines advantages of existing models in a novel way.

The main challenge in the implementation of functional logic languages is the handling of non-determinism. For instance, consider the following operations (in example programs we use Curry syntax [21] which is close to Haskell):

```
flip 0 = 1          coin = 0
flip 1 = 0          coin = 1
```

`flip` is a conventional function whereas `coin` is a *non-deterministic operation* [17], an important concept of contemporary functional logic languages. A non-deterministic operation might yield more than one result on the same input, e.g., `coin` has values `0` and `1` (see [7,17] for discussions of this concept). Due to the importance of non-deterministic operations, Curry defines an archetypal *choice* operation "`?`" by

```
x ? _ = x
_ ? y = y
```

so that one can define `coin` also by "`coin = 0 ? 1`". In functional logic languages, non-deterministic operations can be used as any other operation, in particular, as arguments to other (deterministic) operations, e.g., as in "`flip coin`". It is important to keep in mind that any evaluation of an expression might lead to a non-deterministic choice. We review some existing approaches to deal with such choices during program execution.

Backtracking implements a choice by selecting one alternative to proceed the computation. If a computation comes to an end (failure or success), the state before the choice is restored and the next alternative is taken. Backtracking is the traditional approach of Prolog systems so that it is used in implementations that compile functional logic languages into Prolog, like PAKCS [5,19] or TOY [25]. The major disadvantage of backtracking is its operational incompleteness: if the first alternative does not terminate, no result will be computed.

Copying or *cloning* avoids this disadvantage by copying the context of a choice and proceed with both alternatives in parallel or by interleaving steps. Due to the cost of copying when a choice occurs deeply in an expression, it has been used only in experimental implementations, e.g., [10].

Pull-tabbing is another approach to avoid the incompleteness of backtracking by keeping all alternatives in one computation structure, typically, a graph. It was first sketched in [2] and formally explored in [3]. In contrast to copying, a pull-tab step is a local transformation which moves a choice occurring in an argument of an operation outside this operation. For instance,

```
flip (0 ? 1)  →  (flip 0) ? (flip 1)
```

is a pull-tab step. Pull-tabbing is used in implementations targeting complete search strategies, e.g., KiCS [15], KiCS2 [13], or Sprite [11]. Although pull-tab steps have local effects only, iterated pull-tab steps move choices to the root of an expression. If expressions with choices are shared (e.g., by `let` expressions or multiple occurrences of argument variables in rule bodies), pull-tab steps might produce multiple copies of the same choice. This could lead to unsoundness, which can be fixed by attaching identifiers to choices [3], and to duplication of computations. The latter is a serious problem of pull-tabbing implementations [18]. In this paper, we propose a working solution to this problem by adding a kind of memoization to pull-tabbing. With this extension, pull-tabbing becomes faster than backtracking and at the same time flexible and operationally complete search strategies can be used.

This paper is structured as follows. After reviewing some details of functional logic programming and the pull-tab strategy along with its performance issues in the following two sections, we present our solution to these problems in Sect. 4. A prototypical implementation of our improved strategy is sketched in Sect. 5 and evaluated by some benchmarks in Sect. 6. Related work is discussed in Sect. 7 before we conclude.

2 Functional Logic Programming with Curry

The declarative multi-paradigm language Curry [21], considered in this paper for concrete examples, combines features from functional programming (demand-driven evaluation, parametric polymorphism, higher-order functions) and logic programming (computing with partial information, unification, constraints). The syntax of Curry is close to Haskell[1] [27]. In addition, Curry allows free (logic) variables in conditions and right-hand sides of defining rules. The operational semantics is based on an optimal evaluation strategy [4]—a conservative extension of lazy functional programming and logic programming.

A Curry program consists of the definition of data types (introducing data *constructors*) and *functions* or *operations* on these types. For instance, the data types for Boolean values and polymorphic lists are as follows:

```
data Bool = False | True
data List a = [] | a : List a    -- [a] denotes "List a"
```

A *value* is an expression without defined operations. As mentioned in Sect. 1, Curry allows the definition of non-deterministic operations with the choice operator "`?`" so that the expression "`True ? False`" has two values: `True` and `False`. Using non-deterministic operations as arguments might cause a semantical ambiguity which has to be fixed. For instance, consider the operations

```
xor True  x = not x            not True  = False
xor False x = x                not False = True

xorSelf x = xor x x            aBool = True ? False
```

and the expression "`xorSelf aBool`". If we interpret this program as a term rewriting system, we could have the derivation

```
xorSelf aBool  →  xor aBool aBool  →  xor True aBool
               →  xor True False   →  not False     →  True
```

leading to the unintended result `True`. Note that this result cannot be obtained if we use a strict strategy where arguments are evaluated prior to the function calls. In order to avoid dependencies on the evaluation strategies and exclude such unintended results, González-Moreno et al. [17] proposed the rewriting logic CRWL as a logical (execution- and strategy-independent) foundation for declarative programming with non-strict and non-deterministic operations. CRWL specifies the *call-time choice* semantics [22], where values of the arguments of an operation are determined before the operation is evaluated. This can be enforced in a lazy strategy by sharing actual arguments. For instance, the expression above can be lazily evaluated provided that all occurrences of `aBool` are shared so that all of them reduce either to `True` or to `False` consistently. Thus, sharing is not an option to support an efficient evaluation, but it is required for semantical reasons.

[1] Variables and function names usually start with lowercase letters and the names of type and data constructors start with an uppercase letter. The application of f to e is denoted by juxtaposition ("$f\ e$").

Fortunately, sharing is not an extra burden but already provided by implementations of lazy languages in order to avoid duplication of work. To avoid re-evaluations of identical subexpressions, e.g., the subexpression $f\ e$ in `xorSelf (`$f\ e$`)` where f might cause an expensive computation, the two occurrences of `x` in the right-hand side of the `xorSelf` rule are *shared*. This can be achieved by a graph represention of expressions so that all occurrences of `x` refer to the same graph node. Hence, if $f\ e$ is evaluated as the first argument of `xor` to some value v, the node containing f is replaced by v so that the second argument of `xor` also refers to v. This "update-in-place" of evaluated function calls is essential for lazy languages and also required to ensure the optimality of lazy strategies for functional logic languages [4].

Formally, we can consider programs as graph rewriting systems [28] so that rewrite steps are graph replacements. In order to simplify our presentation, we use the idea to represent sharing by `let` expressions, similarly to Lauchbury's natural semantics for lazy evaluation [23]. This is also used to specify the operational semantics of functional logic languages [1]. Bindings of `let` expressions are stored in a heap so that updates of function nodes are represented as heap updates. Instead of repeating the details of [1], we show the possible evaluations of `xorSelf aBool` in this heap model. Here, the heap is shown on the left, evaluations with the same heap are written in the same line, and new evaluation tasks caused by non-deterministic choices are indented:

```
[]                      let x = aBool in xorSelf x          (1)
[x ↦ aBool]             → xorSelf x → xor x x               (2)
[x ↦ True ? False]      → xor x x                           (3)
  [x ↦ True]              xor x x → not x → [False]         (4)
  [x ↦ False]             xor x x → x → [False]             (5)
```

In line (2), the `let` binding is moved into the heap. The function call in this binding is evaluated and updated in (3). Since this is a choice, a new evaluation task is established for each alternative. Thanks to the sharing of the value of `x`, the unintended value `True` is not computed as a result. Since a heap can be considered as another representation of a graph structure, we use heaps and graphs interchangeably.

3 Pull-Tabbing

If non-deterministic choices are implemented by backtracking, as in Prolog, one has to reason about the influence of the search strategy to the success of an evaluation—a non-trivial task in the presence of a lazy evaluation strategy. To support better search strategies, like breadth-first or parallel search, all non-deterministic choices should be represented in a single (graph) structure so that one can easily switch between different computation branches. As discussed in Sect. 1, pull-tabbing [2,3] has been used in implementations supporting advanced search strategies. A *pull-tab step* moves a choice occurring in a demanded argu-

ment of an operation outside this operation: if f demands the value of its single argument, then

$$f\ (e_1\ \texttt{?}\ e_2) \quad \to \quad (f\ e_1)\ \texttt{?}\ (f\ e_2)$$

is a pull-tab step.

The nice aspects of pull-tabbing are its operational completeness [3] and the locality of steps. Iterated pull-tab steps move choices towards the root:

```
not (not (True ? False)) → not ((not True) ? (not False))
                         → (not (not True)) ? (not (not False))
```

A choice at the root of an expression leads to two new expressions that must be evaluated and might lead to two result values. Since pull-tabbing does not fix some search strategy, it is assumed that these alternative expressions are evaluated by different computation *tasks*. Conceptually, an entire computation consists of a set of tasks where each task evaluates some node of a graph. In this example, the new tasks evaluate the expressions `not (not True)` and `not (not False)`, respectively.

Pull-tab steps as described so far are not sufficient to correctly implement a non-strict functional logic language like Curry. As discussed in Sect. 2, the call-time choice semantics requires to share the values of non-deterministic arguments in a computation. We can implement this requirement by adding identifiers to choices and associating a "fingerprint" [11] to each task. A *fingerprint* is a (partial) mapping from choice identifiers to choice alternatives (*L*eft or *R*ight). When a task reaches a choice at the root, it proceeds as follows:

- If the fingerprint of the task contains a selection for this choice, select the corresponding branch of this choice.
- Otherwise, create two new tasks for the left and right alternative where the fingerprint is extended for this choice with L and R, respectively.

With this refinement of pull-tabbing, we obtain the following evaluation of a variation of "`xorSelf aBool`" (where the fingerprint of the task is written on the left and the heap, which is always $[\texttt{x} \mapsto \texttt{True ?}_1\ \texttt{False}]$, is omitted):

```
[]      xor x x → (xor True x) ?1 (xor False x)
[1/L]     xor True x → not x → (not True) ?1 (not False)
[1/L]     → not True → [False]
[1/R]     xor False x → x → True ?1 False
[1/R]     → [False]
```

Thanks to fingerprints, only values which are correct w.r.t. the call-time choice semantics are produced [3].

Unfortunately, pull-tabbing has some performance problems. In contrast to backtracking, where a non-deterministic choice is implemented by selecting one branch and proceed with this selection until failure or success, pull-tabbing moves each non-deterministic choice up to the root of the expression under evaluation. Hence, the consistency of choices is checked only for choices at the root, i.e., outside function calls. This has the operational consequence that *each* access to

a non-deterministic expression leads to a stepwise shifting of choices towards the root. Thus, multiple accesses to a same non-deterministic expression multiplies the execution time. For instance, consider a function

$$f\ x\ \texttt{=}\ C[x,\ldots,x] \tag{1}$$

where the right-hand side is (or evaluates to) an expression containing n occurrences of the argument x (represented by the context C). Now consider the evaluation of $f\ (e_1\ \texttt{?}\ e_2)$. Whenever some occurrence of x in C is demanded in this evaluation, the choice occurring in the actual argument is moved up to the root by pull-tabbing. Hence, if all n occurrences of x in C are demanded at depth d, approximately $n \cdot d$ pull-tab steps are performed (and most of the resulting choice nodes are omitted at the end due to fingerprinting). In contrast, backtracking is more efficient since it selects one alternative for the first occurrence of x and then simply uses this alternative for all subsequent occurrences of x.

4 Memoized Pull-Tabbing

In this section we present an improvement of pull-tabbing which avoids the performance problems discussed above.

4.1 The Basic Scheme

In principle, the duplication of choices is necessary due to shared subexpressions which are evaluated by different tasks. In contrast to purely functional programming, it would be wrong to update graph nodes of such expressions by their results if they occur in a non-deterministic context. As a solution to this problem, we propose to store "task-specific" updates in the graph, i.e., instead of updating graph nodes by their computed results, we keep the graph nodes but memorize results already computed by some task for a function node. When a task has to evaluate a function node again due to sharing, it directly uses an already computed result.

In order to implement this idea, each task evaluating some expression (subgraph) has a unique identifier (e.g., a number), also called *task identifier*. To store task-specific updates, each graph node representing a function call contains a (partial) map tr, called *task result map*, from task identifiers to graph nodes.

To avoid repeated pull-tab steps, pull-tab steps are not performed for choices that already contain a selection in the fingerprint of the task. In this case, we proceed with the selected branch but have to remember, by using the task result map, that computed results are valid only for this branch. To be more precise, consider a node n representing some function call $f\ (e_1\ \texttt{?}_c\ e_2)$, where f demands its argument. This node is evaluated by the task with identifier i as follows:

- If task i contains no selection for c in its fingerprint, a standard pull-tab step is performed, i.e., two new nodes $n_1 = f\ e_1$ and $n_2 = f\ e_2$ are created and n is updated to $n_1\ \texttt{?}_c\ n_2$.

– If task i contains a selection for c, say L, it would be wrong to update node n to $f\ e_1$ due to possible sharing of n. Instead, a new node $n' = f\ e_1$ is created and $n.tr$ is updated with $n.tr(i) = n'$.

This strategy has the consequence that only the first occurrence of a choice in a computation is moved to the root by iterated pull-tab steps. Since a choice at the root causes a splitting of the current task into two new tasks evaluating the left and right alternative, respectively, this choice, when evaluated again due to sharing, has a selection in the fingerprint so that this selection is immediately taken and stored in the task result map.

Since function calls can be nested, the task result map must be considered for any function call, i.e., also those without a choice in an argument. Thus, when a task with identifier i evaluates some function node n, it checks whether $n.tr(i)$ is defined:

– If $n.tr(i) = n'$, then n' is evaluated instead of n.
– If $n.tr(i)$ is undefined, n is evaluated to some node n' and $n.tr$ is updated with $n.tr(i) = n'$.

Hence, if node n is shared so that task i has to evaluate n again, the already evaluated result is taken.

This evaluation scheme requires a bit more time when function nodes are accessed but avoids the expensive duplication of non-deterministic computations with pure pull-tabbing. We call this improved strategy *memoized pull-tabbing* (*MPT*).

Memoized pull-tabbing can reduce the complexity of non-deterministic computations. For instance, consider again function f defined by rule (1) in Sect. 3. When the expression `let x =` e_1 `?1` e_2 `in` f `x` is evaluated, rule (1) is applied and eventually the first occurrence of `x` is evaluated by a pull-tab step. This leads (by iterated pull-tab steps) to the expression

`let x =` e_1 `?1` e_2 `in` $C[e_1,\mathtt{x},\ldots,\mathtt{x}]$ `?1` $C[e_2,\mathtt{x},\ldots,\mathtt{x}]$

The left and right alternative are further evaluated by two tasks T_1 and T_2 having an L and R selection for choice 1, respectively. Task T_1 evaluates all further occurrences of `x` by selecting e_1 and setting the task result maps of the parent nodes to the results computed with this selection. Hence, instead of $n \cdot d$ pull-tab steps with pure pull-tabbing, MPT performs only d pull-tab steps and $n-1$ "selection" steps for each task T_1 and T_2 to evaluate the initial expression.

Before presenting and evaluating an implementation of MPT, we propose some refinements which will lead to our final MPT strategy.

4.2 Refinements for Deterministic Computations

In typical application programs, large parts of an evaluation are *deterministic computations*, i.e., computations where choice nodes do not occur. Similarly, a *deterministic expression* is an expression whose evaluation does not demand the evaluation of a choice. Since a reasonable implementation of a functional logic

language should support efficient deterministic computations, we present two improvements of our basic MPT strategy for this purpose.

Our first refinement tries to avoid the use of the task result map tr in nodes whenever it is not necessary, in particular, for deterministic computations. For this purpose, each graph node n has an *owner task* (ot), i.e., $n.ot$ is the identifier of the task that created this node. For the initial expression, the owner task of all nodes is the identifier of the main task. When a rule is applied, i.e., a function call is replaced by the right-hand side of a program rule, the owner task of the nodes created for the right-hand side is identical to the owner task of the root of the left-hand side. In case of a pull-tab step

$$f\ (e_1\ \texttt{?}\ e_2) \quad \rightarrow \quad (f\ e_1)\ \texttt{?}\ (f\ e_2)$$

the owner tasks of the new function calls $f\ e_1$ and $f\ e_2$ are the identifiers of the new tasks that will evaluate the left and right alternative, respectively.

In order to compare the owner tasks of nodes, we assume a partial ordering on task identifiers. Note that new tasks are created when a choice appears at the root. In this case the current task t is split into two new tasks t_1 and t_2 which evaluate the left and right alternative of the choice, respectively. We call t *parent* of t_1 and t_2. If i, i_1, i_2 are the identifiers of t, t_1, t_2, respectively, then we assume that $i < i_1$ and $i < i_2$. We call node n_1 *younger* than n_2 if $n_1.ot > n_2.ot$.

If the current task evaluates some choice $n = e_1\ \texttt{?}_c\ e_2$ and the fingerprint of the task already contains a decision for this choice, we follow this decision instead of pushing the choice towards the root by a pull-tab step, as described above. Now we refine the basic scheme by considering the owner tasks. Assume that i is the identifier of the current task, its fingerprint selects e_1 for choice c, and there is the parent node $n' = f\ n$ (for simplicity, we consider only unary functions in this discussion). We distinguish the following cases:

1. If $i > n'.ot$, then $n'.tr(i)$ is set to a new node $n'' = f\ e_1$ with $n''.ot = i$ and the evaluation proceeds with node n''.
2. Otherwise ($i = n'.ot$), node n' is updated in place such that $n' = f\ e_1$.

Next consider the situation that there is some function node $n = f\ a$ and the argument a has been evaluated to a'.

1. If a' is younger than n, the argument has been evaluated to some value which is valid only in the new task which created a'. Instead of updating n in place, $n.tr(a'.ot)$ is set to a new node $n' = f\ a'$ with $n'.ot = a'.ot$ and the evaluation proceeds with node n' instead of n.
2. Otherwise (a' is not younger than n), the value computed for a is valid for n so that n is updated in place such that $n = f\ a'$.

Thus, for deterministic computations, which are performed in a single task (tasks are created only for non-deterministic steps), the task result maps are not used at all.

A further refinement exploits the tree structure of tasks. An *ancestor* of a task is either its parent or the parent of some ancestor of the task. Consider the case that we have to evaluate some function node n in a task identified by i:

1. If $n.tr(i)$ is defined, then task i already evaluated node n so that we can proceed with $n.tr(i)$ instead of n.
2. If $n.tr(i)$ is not defined and j is a parent of i so that $n.tr(j) = n'$, then n' is also a valid result of n so that we can proceed with n'. Hence, we follow the ancestor chain of i to find the first ancestor k such that $n.tr(k)$ is defined.
3. Otherwise (there is no ancestor j of i with $n.tr(j)$ defined), we evaluate n.

Note that the owner task of nodes changes in a computation only if some choice node is evaluated. In case of a pull-tab step, the new nodes are younger than the choice node. If there is some decision for the choice w.r.t. the fingerprint of the current task, we commit to the selected branch and set the owner task of this selection to the current task. An interesting consequence of this strategy is that the owner tasks of nodes in a computation are not changed when choice nodes do not occur in this computation. In particular, deterministic computations without occurrences of choice nodes are always evaluated in place—independently of the task which evaluated them. This has the effect that deterministic expressions are evaluated at most once, even if they are shared among non-deterministic branches. This property, also called *sharing across non-determinism* [15], is an important feature of the pull-tab strategy. Consider an expression

`let` x `=` e `in` $C_1[x]$ `?` $C_2[x]$

where e is a deterministic expression and the value of x is demanded in both $C_1[x]$ and $C_2[x]$. Then e will be evaluated only once since the task evaluating $C_1[x]$ will replace e by its result so that this result is available for the task evaluating $C_2[x]$. In contrast, an implementation based on backtracking would evaluate e two times since the evaluation of e by the task evaluating $C_1[x]$ will be undone before evaluating $C_2[x]$. Note that this is not only a problem of backtracking. For instance, the approach to implement call-time choice with purely functional programming features presented in [16] also reports the lack of sharing across non-determinism.

5 Implementation

In order to evaluate our ideas, we implemented MPT in Julia[2], a high-level dynamic programming language. We used Julia due to its direct support of dynamic data structures, garbage collection, and higher-order features. By exploiting the intermediate language ICurry [9], the compiler from ICurry to Julia is approximately 300 lines of Curry code. Furthermore, the run-time system, responsible to implement the computation graph, pull-tab steps, computation tasks with various search strategies, and some more aspects, consists of approximately 300 lines of Julia code. Thus, this implementation, called "JuCS" (Julia Curry System[3]), is a proof of concept which could also be implemented, with more effort and probably more efficiently, in other imperative languages, like C. In the following, we sketch some aspects of this implementation.

[2] `https://julialang.org/`
[3] Available at `https://github.com/cau-placc/julia-curry`

Apart from the memoized pull-tabbing strategy, the implementation has many similarities to Sprite [11]. Expressions are represented as a graph structure. To distinguish different kinds of graph nodes (function, constructor, choice, failure, etc), each node has a tag. Furthermore, a node contains an integer value (choice identifier, constructor index, etc), the identifier of the owner task, an array of references to argument nodes, a code reference in case of function nodes, and a task result map (a Julia dictionary with task identifiers as keys and node references as values). The run-time system works on a queue of tasks where each task contains a unique number, the root node evaluated by this task, the fingerprint, and the identifiers of the parent tasks. With these data structures, the run-time system evaluates expressions, as described above, by computing the head normal form of the root node of the current task. If this yields a choice node, two new tasks are created with extended fingerprints. In case of depth-first search, these tasks are added at the front of the task queue, while they are added at the back in case of breadth-first search.

Free variables and their bindings require a non-trivial implementation with non-deterministic value generator operations in a pure pull-tabbing implementation [14]. Our MPT strategy allows a much simpler implementation. Instead of representing free variables as value generators (as in [6]), JuCS has a "free" tag for nodes where the task result map is used to store task-specific bindings for free variables. Hence, free variables are handled as efficient as in Prolog implementations while still allowing more flexible search strategies.

In order to compare the different run-time models (MPT, pull-tabbing, backtracking) inside our implementation, JuCS contains also two alternative run-time systems implementing pure pull-tabbing and backtracking. The pull-tabbing system is a reduced variant of the standard run-time system. The backtracking system uses ideas from Prolog implementations, in particular, the improved backtracking and trailing mechanism of Warren's Abstract Machine [29] to reduce the amount of stored backtrack information.

6 Benchmarks

Memoized pull-tabbing requires more effort at run time than pure pull-tabbing due to the tests when evaluating or updating a function node in the computation graph. Thus, it is interesting to see whether this pays off in practice. Therefore, we executed a set of benchmarks with our new implementation and compared the execution times[4] of the different run-time systems provided with this implementation. The results are summarized in Table 1.

The first two examples[5] are purely deterministic programs: `nrev` is the quadratic naive reverse algorithm on a list with 4096 elements and `takPeano` is a highly recursive function on naturals [26] applied to arguments (24,16,8), where numbers and arithmetic operations are in Peano representation. `addNum2` and

[4] All benchmarks were executed on a Linux machine running Ubuntu 18.04 with an Intel Core i7-85550U (1.8GHz) processor.

[5] The actual programs are available with the implementation described in Sect. 5.

Program	MPT	pull-tab	backtrack
`nrev`	2.37 s	2.29 s	7.09 s
`takPeano`	12.04 s	11.84 s	31.78 s
`addNum2`	0.46 s	6.21 s	0.39 s
`addNum5`	1.61 s	47.69 s	1.26 s
`select50`	0.05 s	4.18 s	0.14 s
`select75`	0.10 s	24.10 s	0.31 s
`select100`	0.18 s	111.47 s	0.55 s

Table 1. Evaluating different run-time systems of JuCS

Program	MPT	pull-tab	backtrack
`sort1`	9.83 s	9.96 s	15.47 s
`sort2`	9.84 s	9.73 s	155.64 s

Table 2. Effect of sharing across non-determinism

`addNum5` non-deterministically choose a number (out of 2000) and add it two and five times, respectively. `select`n non-deterministically selects an element in a list of length n and sums up the element and the list without the the selected element.

As one can see from the direct comparison to pure pull-tabbing, the overhead caused by the additional checks required for memoization is limited and immediately pays off when non-deterministic expressions are shared, which is often the case in applications involving non-determinism.

It is interesting to note that the backtracking strategy is less efficient than MPT, although MPT supports more flexible search strategies. This might be due to the fact that backtracking has to check, for each reduction step, whether this step has to be remembered in order to undo it in case of backtracking.

As already discussed, backtracking has another disadvantage: deterministic computations are not shared across non-deterministic computations. This has the unfortunate consequence that a client using some algorithm of a library has to know whether this algorithm is implemented with non-deterministic features, since arguments might be evaluated multiple times with backtracking. To show this effect, consider the deterministic insertion sort operation `isort`, the non-deterministic permutation sort operation `psort`, and the infinite list of all prime numbers `primes` together with the following definitions (this example is inspired from [13]):

```
sort1 = isort [primes!!303, primes!!302, primes!!301, primes!!300]
sort2 = psort [primes!!303, primes!!302, primes!!301, primes!!300]
```

In principle, one would expect that the execution time of `sort1` is almost equal to the time to execute `sort2` since the time to sort a four-element list is neglectable. However, implementations based on backtracking evaluate the primes occurring in `sort2` multiple times, as can be seen by the run times shown in Table 2.

Example	KiCS2 (BFS/DFS)	JuCS (BFS/DFS)
`addNum2`	2.18	1.02
`addNum5`	1.58	1.00
`select50`	1.03	1.00
`select100`	1.28	1.04
`permsort`	4.46	1.11

Table 3. Relative execution times of BFS vs. DFS

We already emphasized the fact that pull-tabbing supports flexible search strategies. Since all non-deterministic values of an expression are represented in one structure, different search strategies can be implemented as traversals on this structure. For instance, KiCS2 evaluates each expression to a tree of its values so that the top-level computation collecting all values can be defined as a traversal on this tree [13]. Our implementation uses a queue of tasks so that search strategies can be implemented as specific strategies to put and get tasks to and from the queue, respectively (as sketched above). The practical behavior of search strategies in KiCS2 was analyzed in [20] (since then, breadth-first search became the default strategy for KiCS2). Table 3 shows that MPT has an even better behavior since, in contrast to pure pull-tabbing, it is not necessary to move all choices to the root in order to build a tree of all values. Here, we also added the classical permutation sort example since it showed a larger slowdown of breadth-first search in [20].

7 Related Work

In this section we review other approaches to implement functional logic languages and relate our proposal to them.

Early approaches to implement functional logic languages exploited Prolog's backtracking strategy for non-deterministic computations [5,24]. By adding a mechanism to implement demand-driven evaluation, one can use Prolog as a target language, as done in PAKCS [19] and TOY [25]. The usage of Prolog yields also a direct support for free variables. However, such implementations suffer from the operational incompleteness of the backtracking strategy.

Pull-tabbing supports more flexible search strategies by representing choices as data. The theoretical properties are investigated in [3]. On the practical side, pull-tabbing is useful to implement non-determinism in a deterministic target language. For instance, ViaLOIS [12] uses pull-tabbing to translate Curry programs into Haskell and OCaml programs, respectively. ICurry [9] is an intermediate language intended to translate Curry programs into imperative target languages. It has been used to translate Curry to LLVM code [11] and to C or Python programs [30]. The operational semantics of ICurry is specified in [9] by an abstract machine which performs pull-tab steps and uses a graph structure to represent expressions with sharing and a queue of tasks, where each task has its own fingerprint to implement the selection of consistent choices.

The Curry compiler KiCS2 [13] is based on pull-tabbing and compiles Curry programs into a purely functional Haskell programs. Non-determinism is implemented by representing choices as data terms so that Curry expressions are evaluated to choice trees. Pull-tab steps are encoded as rules for choice terms. Values are extracted from choice trees by traversing them with fingerprints. Hence, KiCS2 implements non-determinism in a modular way: any expression is evaluated to a choice tree representation of all its values, and there is a separate operation which extracts correct values from the choice tree structure. Therefore, KiCS2 implements various search strategies as different tree traversal strategies, and infinite search spaces (choice trees) do not cause problems due to Haskell's lazy evaluation strategy.

Since KiCS2 suffers from the performance problems of pure pull-tabbing, an eager evaluation of demanded non-deterministic subexpressions is proposed in [18]. An automatic program transformation implementing this optimization is based on a demand analysis. However, this approach does not work for arbitrary programs since a precise demand analysis for complex data structures is non-trivial and not yet available for functional logic programs. Therefore, it is an interesting question for future work whether our MPT scheme can be combined with the purely functional implementation approach of KiCS2.

Sharing across non-determinism describes the property that deterministic subexpressions shared in different non-deterministic branches are evaluated at most once. This is usually not the case in implementations based on backtracking. As emphasized in [15], pull-tabbing easily ensures this property if the target language implements sharing for common subexpressions, as in our implementation or in lazy functional target languages [13,15].

An approach to support functional logic programming features as in Curry in a purely functional language is a library for non-deterministic computations with (explicit) sharing [16]. The key idea of this library is to translate non-deterministic computations into monadic computations that manipulate a *thunk store*. The thunk store holds either unevaluated computations or their results, which may again contain unevaluated arguments, and is closely related to the heap described in Sect. 2. The library provides an explicit `share` operation to allow the sharing of computations. Shared computations are initially entered unevaluated into the thunk store and only the demand for a computation triggers its evaluation. If a computation is non-deterministic, the thunk store is updated with the corresponding result independently in each branch. All subsequent uses of the shared computation within one computation branch then reuse the updated result in the thunk store. Although shared results are reused in one computation branch if demanded more than once, the library does not support sharing across non-determinism because shared computations are evaluated independently in different branches. Due to the fact that the implementation relies on the type class `MonadPlus`, different search strategies can be exploited depending on the concrete instance of `MonadPlus`. Furthermore, the library has no direct support of free variables and can only emulate them by using non-deterministic generators [6].

	Back-tracking	Explicit Sharing	Pull Tabbing	MPT
Flexible search strategies	−	+	+	+
Free variables	+	−	−	+
Sharing across non-determinism	−	−	+	+
Sharing non-determinism	+	+	−	+

Table 4. Comparing properties of implementation strategies

Table 4 compares the properties of the various approaches to implement demand-driven non-deterministic computations discussed above. "Flexible search strategies" means whether only one or a number of different search strategies are supported. "Free variables" denotes a direct support of free variables. This is not the case for explicit sharing and pull-tabbing, since they require a simulation of free variables by non-deterministic generator operations and non-trivial techniques to obtain the effect of binding free variables through unification [14]. "Sharing across non-determinism" describes the aforementioned ability to reuse already computed results of deterministic subexpressions in different branches of non-deterministic computations. "Sharing non-determinism" means that the results of already evaluated non-deterministic subexpressions are re-used when these subexpressions are shared. As one can see, our new MPT strategy is the only implementation which combines all these properties. As shown by our benchmarks, this has a positive effect on the efficiency of MPT on a range of different application scenarios.

8 Conclusions

The efficient implementation of functional logic programming languages is still a challenge due to the combination of advanced declarative programming concepts. In order to free the programmer from considering details about the concrete evaluation strategy, it is desirable to support operationally complete strategies which ensure that values are computed whenever they exist. This can be obtained by representing the complete state with all branches of a non-deterministic computation in one data structure. Pull-tabbing is a simple and local transformation to deal with non-deterministic choices. However, pull-tabbing has the risk to duplicate work during evaluation. In this paper we proposed a significant improvement by adding a kind of memoization to pull-tabbing. As we demonstrated by our benchmarks, this improved evaluation mechanism does not cause much overhead, is often faster than backtracking, and can dramatically improve pure pull-tabbing. Morever, it keeps all the positive properties of pull-tabbing: application of various search strategies and sharing across non-determinism. Our prototypical implementation showed that it can also be implemented with modest efforts: Curry programs can be compiled by using the already existing intermediate language ICurry in a straightforward manner, and the run-time system is

quite compact. Thus, it is an ideal model to implement multi-paradigm declarative languages also with other target languages, e.g., to integrate declarative programming in applications written in other imperative languages.

Nevertheless, there is room for future work. For instance, one could try to identify non-shared subexpressions, e.g., by some sharing or linearity analysis, to avoid run-time checking of memoized data. Another interesting question is whether it is possible to implement the presented ideas in a purely functional manner so that one can use them to improve existing approaches like KiCS2 [14] or the library for explicit sharing [16].

References

1. E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. Operational semantics for declarative multi-paradigm languages. *Journal of Symbolic Computation*, 40(1):795–829, 2005.
2. A. Alqaddoumi, S. Antoy, S. Fischer, and F. Reck. The pull-tab transformation. In *Proc. of the Third International Workshop on Graph Computation Models*, pages 127–132. Enschede, The Netherlands, 2010. Available at http://gcm2010.imag.fr/pages/gcm2010-preproceedings.pdf.
3. S. Antoy. On the correctness of pull-tabbing. *Theory and Practice of Logic Programming*, 11(4-5):713–730, 2011.
4. S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *Journal of the ACM*, 47(4):776–822, 2000.
5. S. Antoy and M. Hanus. Compiling multi-paradigm declarative programs into Prolog. In *Proc. International Workshop on Frontiers of Combining Systems (FroCoS'2000)*, pages 171–185. Springer LNCS 1794, 2000.
6. S. Antoy and M. Hanus. Overlapping rules and logic variables in functional logic programs. In *Proceedings of the 22nd International Conference on Logic Programming (ICLP 2006)*, pages 87–101. Springer LNCS 4079, 2006.
7. S. Antoy and M. Hanus. Functional logic programming. *Communications of the ACM*, 53(4):74–85, 2010.
8. S. Antoy and M. Hanus. Contracts and specifications for functional logic programming. In *Proc. of the 14th International Symposium on Practical Aspects of Declarative Languages (PADL 2012)*, pages 33–47. Springer LNCS 7149, 2012.
9. S. Antoy, M. Hanus, A. Jost, and S. Libby. ICurry. In *Declarative Programming and Knowledge Management - Conference on Declarative Programming (DECLARE 2019)*, pages 286–307. Springer LNCS 12057, 2020.
10. S. Antoy, M. Hanus, J. Liu, and A. Tolmach. A virtual machine for functional logic computations. In *Proc. of the 16th International Workshop on Implementation and Application of Functional Languages (IFL 2004)*, pages 108–125. Springer LNCS 3474, 2005.
11. S. Antoy and A. Jost. A new functional-logic compiler for curry: Sprite. In *Proceedings of the 26th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2016)*, pages 97–113. Springer LNCS 10184, 2016.
12. S. Antoy and A. Peters. Compiling a functional logic language: The basic scheme. In *Proc. of the Eleventh International Symposium on Functional and Logic Programming*, pages 17–31. Springer LNCS 7294, 2012.

13. B. Braßel, M. Hanus, B. Peemöller, and F. Reck. KiCS2: A new compiler from Curry to Haskell. In *Proc. of the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011)*, pages 1–18. Springer LNCS 6816, 2011.
14. B. Braßel, M. Hanus, B. Peemöller, and F. Reck. Implementing equational constraints in a functional language. In *Proc. of the 15th International Symposium on Practical Aspects of Declarative Languages (PADL 2013)*, pages 125–140. Springer LNCS 7752, 2013.
15. B. Braßel and F. Huch. On a tighter integration of functional and logic programming. In *Proc. APLAS 2007*, pages 122–138. Springer LNCS 4807, 2007.
16. S. Fischer, O. Kiselyov, and C. Shan. Purely functional lazy nondeterministic programming. *Journal of Functional programming*, 21(4&5):413–465, 2011.
17. J.C. González-Moreno, M.T. Hortalá-González, F.J. López-Fraguas, and M. Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming*, 40:47–87, 1999.
18. M. Hanus. Improving lazy non-deterministic computations by demand analysis. In *Technical Communications of the 28th International Conference on Logic Programming*, volume 17, pages 130–143. Leibniz International Proceedings in Informatics (LIPIcs), 2012.
19. M. Hanus, S. Antoy, B. Braßel, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, F. Steiner, and F. Teegen. PAKCS: The Portland Aachen Kiel Curry System. Available at `http://www.informatik.uni-kiel.de/~pakcs/`, 2018.
20. M. Hanus, B. Peemöller, and F. Reck. Search strategies for functional logic programming. In *Proc. of the 5th Working Conference on Programming Languages (ATPS'12)*, pages 61–74. Springer LNI 199, 2012.
21. M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.9.0). Available at `http://www.curry-lang.org`, 2016.
22. H. Hussmann. Nondeterministic algebraic specifications and nonconfluent term rewriting. *Journal of Logic Programming*, 12:237–255, 1992.
23. J. Launchbury. A natural semantics for lazy evaluation. In *Proc. 20th ACM Symposium on Principles of Programming Languages (POPL'93)*, pages 144–154. ACM Press, 1993.
24. R. Loogen, F. López Fraguas, and M. Rodríguez Artalejo. A demand driven computation strategy for lazy narrowing. In *Proc. of the 5th International Symposium on Programming Language Implementation and Logic Programming*, pages 184–200. Springer LNCS 714, 1993.
25. F. López-Fraguas and J. Sánchez-Hernández. TOY: A multiparadigm declarative system. In *Proc. of RTA'99*, pages 244–247. Springer LNCS 1631, 1999.
26. W. Partain. The nofib benchmark suite of Haskell programs. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, pages 195–202. Springer, 1993.
27. S. Peyton Jones, editor. *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press, 2003.
28. D. Plump. Term graph rewriting. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages and Tools*, pages 3–61. World Scientific, 1999.
29. D.H.D. Warren. An abstract Prolog instruction set. Technical note 309, SRI International, Stanford, 1983.
30. M.A. Wittorf. Generic translation of Curry programs into imperative programs (in German). Master's thesis, Kiel University, 2018.