From Non-determinism to Goroutines: A Fair Implementation of Curry in Go

Jonas Böhm
CAU Kiel
Institut für Informatik
Kiel, Germany
boehm-jonas@gmx.de

Michael Hanus
CAU Kiel
Institut für Informatik
Kiel, Germany
mh@informatik.uni-kiel.de

Finn Teegen
CAU Kiel
Institut für Informatik
Kiel, Germany
fte@informatik.uni-kiel.de

ABSTRACT

The declarative programming language Curry amalgamates demand-driven evaluation from functional programming with non-determinism from logic programming. In contrast to Prolog, the search strategy for non-deterministic computations is not fixed so that complete or parallel strategies are reasonable for Curry. In particular, a desirable option is a fair strategy which frees the programmer from considering the influence of the search strategy to the success of a computation. In this paper we describe an implementation with this property. Based on recent developments on operational models for functional logic programming, we present a new implementation which transforms Curry programs in several transformation steps into Go programs. By exploiting lightweight threads in the form of goroutines, we obtain a complete and fair implementation which automatically uses multi-processing to speed up non-deterministic computations. This has the effect that, in some cases, non-deterministic algorithms are more efficiently evaluated than deterministic ones.

CCS CONCEPTS

• **Software and its engineering → Multiparadigm languages**; **Functional languages**; **Constraint and logic languages**; **Compilers**.

KEYWORDS

Declarative programming, implementation, parallelism

ACM Reference Format:
Jonas Böhm, Michael Hanus, and Finn Teegen. 2021. From Non-determinism to Goroutines: A Fair Implementation of Curry in Go. In *23rd International Symposium on Principles and Practice of Declarative Programming (PPDP 2021), September 6–8, 2021, Tallinn, Estonia.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3479394.3479411

1 INTRODUCTION

Declarative programming languages are based on the idea to express properties about a given problem in a high-level, execution-independent manner, e.g., equations in functional programming or implications (Horn formulas) in logic programming. Ideally, the implementation of such languages should ensure that results are computed when they exist. In functional programming, a result is a value (data term) equivalent to an initial expression w.r.t. the given equations. In logic programming, a result is a substitution of the variables occurring in the initial goal so that the goal instantiated with this substitution is a logical consequence of the program. In a combined functional logic language, such as Curry [32], results are values with possibly constraints on the free variables in the initial expression.

The practice of declarative programming is, however, a bit different. For instance, functional programs, e.g., written in Haskell [44], are sets of functions defined by equations, but the meaning of equations depends on a non-trivial pattern matching translator. For instance, consider the Haskell program

```
f x True = 0
f True y = 1
```

Although the second equation might indicate that the value of `f True True` is `1`, Haskell returns only the value `0`, since rules are tried from top to bottom and only the first matching rule is applied. Although one gets at least a result in this case, it can be worse. Consider the function definitions

```
g True True  = 0              loop = loop
g x    False = 1
```

and the expression `g loop False`. The second equation defining `g` indicates that `1` is a value equivalent to this expression. However, Haskell does not return any result due to its pattern matching strategy.

On the other hand, the same definitions interpreted in Curry return the expected values, since Curry is based on a complete and optimal evaluation strategy [4, 6]. In the case of the expression `f True True`, Curry returns both possible values `0` and `1`. Hence, operations can have more than one result on a fixed input. Conceptually, operations have a set-valued semantics and they are often termed *non-deterministic operations* [23], an important concept of contemporary functional logic languages (see [11, 23] for discussions of this concept). As a consequence, Curry interprets equations as rewrite rules which are applied from left to right. In the case of overlapping rules, as the rules for `f` above, all rules are tried in a non-deterministic manner, similarly to the resolution principle of logic programming. This raises the question about the search strategy in the case of non-deterministic steps. Consider the expression `f True loop`. If the first rule of `f` should be applied, it is necessary to evaluate the second argument `loop` which does not terminate. If the second rule of `f` should be applied, it is not necessary to evaluate

PPDP 2021, September 6–8, 2021, Tallinn, Estonia

ACM ISBN 978-1-4503-8689-0/21/09…$15.00
https://doi.org/10.1145/3479394.3479411

`loop` to apply this rule so that we obtain the result `1`. Hence, the order of rule applications might influence the computed results. The logic programming language Prolog uses a backtracking strategy based on a sequential ordering of program rules. It is well known that this causes operational incompleteness, i.e., results might not be computed. Hence, a Prolog programmer has to take into account the influence of the backtracking search strategy on the computed results. Actually, there are implementations of functional logic languages that compile into Prolog, like PAKCS [7, 29] or TOY [40]. Such implementations inherit the operational incompleteness from Prolog.

In this work we are interested in an operationally complete implementation. This frees the programmer from thinking about details of pattern matching or search, thus, supporting a stronger declarative programming view.[1] This also yields a tight integration between specifications and code where (non-deterministic) specifications and more efficient implementations can be written in a single language [12] in order to enable automatic testing [27] or verification [28].

In order to achieve this goal, we develop a new implementation based on recent developments in operational models for functional logic programming. The main aspect of this implementation is to represent all non-deterministic choices in a single graph structure. This is also the basis of other implementations that are intended to support better search strategies rather than backtracking, e.g., KiCS [20], KiCS2 [19], or Sprite [18]. Similarly to implementations of non-strict functional languages, this graph structure is constructed on demand. However, a semantically correct implementation of demand-driven non-deterministic computations requires some additional structures that could lead to a considerably increased complexity of computations compared to backtracking (this will be explained later in more detail). A solution to this problem has been presented in [31] where the graph structure is enriched with information to memoize non-deterministic computations, called *memoized pull-tabbing*. In this paper we present an implementation of these ideas with a run-time system implemented in Go. In particular, we make the following contributions.

- We present a formal specification of memoized pull-tabbing which is the basis of our implementation.
- We show that the model of memoized pull-tabbing enables a flexible choice of various search strategies.
- In our concrete implementation, we exploit "goroutines" (lightweight threads supported in Go) for tasks representing non-deterministic computations. As a result, our implementation is complete and *fair* w.r.t. non-determinism. Fairness means that all the values of an expression are eventually produced, even if some branches of a computation run forever.
- We demonstrate that non-deterministic algorithms are sometimes more efficiently evaluated than deterministic ones. This is due to the fact that, in our implementation, non-deterministic computations automatically exploit several processors, if available, to speed up the evaluation of expressions.

[1] Of course, details about the evaluation strategy might be relevant to analyze the complexity of a concrete execution, but this is outside the scope of this paper.

This paper is structured as follows. The next section reviews some necessary details about functional logic programming and Curry. The intermediate languages used in our implementation are reviewed in Sect. 3 and 4. Section 5 introduces the basic operational mechanism to deal with non-determinism in our implementation. Since this has some drawbacks, we describe in Sect. 6 our operational model and provide a formal specification of it. This is the basis of our implementation which is sketched in Sect. 7. We evaluate it in Sect. 8 with a couple of benchmarks before we discuss related work and conclude.

2 FUNCTIONAL LOGIC PROGRAMMING AND CURRY

This section reviews some aspects of functional logic programming and the multi-paradigm declarative language Curry that are necessary to understand the contents of this paper. More details can be found in surveys on functional logic programming [11, 26] and in the language report [32].

Curry combines features from functional programming (demand-driven evaluation, strong typing with parametric polymorphism, higher-order functions) and logic programming (non-determinism, computing with partial information, constraints). The syntax of Curry is close to Haskell[2] [44] but allows *free* (*logic*) *variables* in conditions and right-hand sides of defining rules. Since free variables can be replaced by non-deterministic generator operations [9], we concentrate in the following on the handling of non-determinism, which is the main implementation challenge compared to purely functional languages. The operational semantics is based on an optimal evaluation strategy [4, 6]—a conservative extension of lazy functional programming and logic programming.

Similarly to functional programming, a Curry program defines 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"
```

Operations are defined by sets of rules as already presented in the introduction. *Expressions* might contain operations, constructors, and variables, whereas a *value* is an expression without defined operations. The objective of a *computation* is to reduce a given expression to some value by applying program rules.

As mentioned in Sect. 1, an important feature of contemporary functional logic languages are *non-deterministic operations* [23], i.e., operations that yield more than one result for a given argument. Curry defines an archetypal *choice* operation "?" by

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

so that the expression "`False ? True`" has two values: `False` and `True`.

[2] 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 ("$f\ e$").

$$
\begin{array}{lcll}
P & ::= & D_1 \ldots D_m & \text{(program)} \\
D & ::= & f(x_1, \ldots, x_n) = e & \text{(function definition)} \\
e & ::= & x & \text{(variable)} \\
 & | & c(e_1, \ldots, e_n) & \text{(constructor call)} \\
 & | & f(e_1, \ldots, e_n) & \text{(function call)} \\
 & | & \mathit{case}\ e\ \mathit{of}\ \{p_1 \to e_1; \ldots; p_n \to e_n\} & \text{(case expression)} \\
 & | & e_1\ \mathit{or}\ e_2 & \text{(disjunction)} \\
 & | & \mathit{let}\ \{x_1 = e_1; \ldots; x_n = e_n\}\ \mathit{in}\ e & \text{(let binding)} \\
 & | & \mathit{let}\ x_1, \ldots, x_n\ \mathit{free\ in}\ e & \text{(free variables)} \\
p & ::= & c(x_1, \ldots, x_n) & \text{(pattern)}
\end{array}
$$

Figure 1: Abstract syntax of the intermediate language FlatCurry

Although non-deterministic operations have a set-valued semantics [23], they return (non-deterministically) individual values rather than sets of values. This has the advantage that a non-deterministic operation can be used as any other operation, in particular, as an argument or in definitions of other operations. For instance, consider the following operation that inserts an element at an unspecified position into a list:

```
insert :: a → [a] → [a]
insert x ys     = x : ys
insert x (y:ys) = y : insert x ys
```

Hence, the expression `insert 0 [1,2]` non-deterministically evaluates to one of the values `[0,1,2]`, `[1,0,2]`, or `[1,2,0]`. Based on this operation, one can easily define permutations:

```
perm :: [a] → [a]
perm []     = []
perm (x:xs) = insert x (perm xs)
```

Although `perm` is defined by non-overlapping rules, the use of `insert` has the effect that `perm [1,2,3,4]` non-deterministically evaluates to all 24 permutations of the input list.

Compared to approaches where sets or lists of values are passed between operations, as in the "list of successes" approach in purely functional programming [48], the use of non-deterministic operations could lead to simpler program structures. Moreover, it has also operational advantages: since expressions are evaluated on demand, non-deterministic operations as arguments result in a demand-driven construction of the search space, leading to considerable smaller search spaces (see [11, 23] for more detailed discussions). Thus, it is important to keep in mind that any evaluation of an expression might lead to a non-deterministic choice between several expressions or values. This demands for specific implementation techniques which will be discussed later.

If non-deterministic operations occur as arguments to be evaluated, there is a semantical ambiguity which has to be fixed in a concrete programming language. To discuss this, consider the operations

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

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

and the expression "`xorSelf aBool`". If we just apply program rules from left to right, as in term rewriting, we could have the derivation

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

The result `True` is unintended for the operation `xorSelf`. This value cannot be obtained with a strict strategy where arguments are evaluated prior to the function calls. To avoid dependencies on the evaluation strategy and exclude such unintended results, González-Moreno et al. [23] 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 [37], 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 `False` or to `True` consistently.

Sharing is used in implementations of non-strict functional languages in order to support optimal evaluation [36], i.e., typical implementations are based on graph rewriting [43]. We will also base our implementation on graph structures and consider programs as graph rewriting systems [45] so that rewrite steps are graph replacements. To support non-deterministic computation steps, we will represent non-deterministic choices as graph nodes. The precise evaluation of arguments is influenced by the patterns in the left-hand side of program rules. We already discussed in Sect. 1 how the pattern matching strategy influences the success of computations. In order to make this strategy explicit and precise, we discuss in the next sections intermediate languages that are relevant for our implementation.

3 FLATCURRY

Curry has many more features than described so far. Since it is conceptually intended as an extension of Haskell (although not all features are included), Curry also supports higher-order features, scoping with local definitions, modules, monadic I/O [49], as well as specific functional logic programming features, like set functions [10] to encapsulate non-deterministic search, functional patterns [8], or default rules [13]. Due to the complexity of the source language, language processing tools for Curry, like compilers, analyzers, optimization or verification tools, often use an

D	::=	$f = blck$	(function definition)
$blck$	::=	$decl_1 \ldots decl_k\ asgn_1 \ldots asgn_n\ stm$	(block)
$decl$	::=	$declare\ x$	(local variable declaration)
$asgn$	::=	$v = exp$	(variable assignment)
stm	::=	$return\ exp$	(return statement)
	|	$exempt$	(failure statement)
	|	$case\ x\ of\ \{c_1 \to blck_1; \ldots; c_n \to blck_n\}$	(case statement)
exp	::=	v	(variable)
	|	$NODE(l, exp_1, \ldots, exp_n)$	(node construction)
	|	$exp_1\ or\ exp_2$	(disjunction)
v	::=	x	(local variable)
	|	$v[i]$	(node access)
	|	$ROOT$	(root of function call)
l	::=	c	(constructor symbol)
	|	f	(function symbol)

Figure 2: Abstract syntax of function definitions in ICurry

intermediate language, called FlatCurry. The operational semantics of Curry is also specified with FlatCurry [2] by extending Launchbury's natural semantics for lazy evaluation [39] with rules to cover non-determinism. Since we will use FlatCurry in our compilation chain, as various other Curry systems (like PAKCS [29], KiCS [20], KiCS2 [19], or Sprite [18]), we sketch its structure.

The abstract syntax of FlatCurry is summarized in Fig. 1. Data declarations are omitted since they are almost identical to the source language. Thus, a FlatCurry program is a sequence of function definitions, where each function is defined by a single rule with a linear left-hand side, i.e., the argument variables $x_1, \ldots, x_n$ are pairwise different. The right-hand side of the definition consists of variables, constructor or function calls, *case* expressions, disjunctions (written with *or* in infix notation), *let* bindings, or introduction of free variables. The patterns p_i in a *case* expression are pairwise different constructors applied to variables. Therefore, complex function definitions with overlapping or deep patterns in source programs are represented by disjunctions and nested *case* expressions.

To omit the syntactic sugar of the source language, various transformations are applied to generate FlatCurry programs. Local definitions are eliminated by lambda lifting [38], patterns are compiled into simple *case* expressions, and overlapping rules are represented by *or* expressions (see [32] for a precise description of these transformations). For instance, the non-deterministic operation `insert` defined above is represented in FlatCurry as

```
insert(x,xs) = (x:xs)
               or
               case xs of { y:ys → y : insert(x,ys) }
```

The operation `g` defined in Sect. 1 is transformed into FlatCurry as

```
g(x,y) = case y of
           { True  → case x of
                       { True → 0 } ;
             False → 1
           }
```

Due to this transformation, the first branch is on the second argument so that Curry returns `1` for the expression `g loop False`. This kind of pattern matching, which is slightly different from Haskell, is the basis of Curry's complete and optimal evaluation strategy [4, 6].

Any Curry program can be translated into a FlatCurry program. There exists a front end, used by various compilers and tools, to process and compile Curry programs into FlatCurry. The front end can also be accessed in Curry programs by a Curry package to represent FlatCurry programs.[3]

4 ICURRY

FlatCurry is useful when Curry is compiled into Prolog [7, 29] or Haskell [19, 20]. However, when compiling Curry into typical imperative languages, like C, C++, Go, or Python, FlatCurry is less appropriate since some constructs cannot be directly mapped into imperative code, as discussed in [15]. This motivated the introduction of another intermediate language, called ICurry, which is the basis of our implementation. In the following, we review the structure of ICurry. More details can be found in [15].

The abstract syntax of operations in ICurry is summarized in Fig. 2 (similarly to FlatCurry, data declarations are omitted). In contrast to FlatCurry, where the body of a function is just an expression, a function definition in ICurry consists of (optional) variable declarations, assignments, and a final statement. The statement either returns an expression, marks a failure (*exempt*), or performs a case distinction on a given variable. This is oriented towards imperative languages where switches or case distinctions are control structures rather than expressions. In contrast to FlatCurry, a *case* in ICurry compares only the tag of the discriminating variable with the tag of the constructors available for the data type (although we write constructor names in examples, ICurry uses unique tags, e.g., numbers, for the constructors). The arguments of the constructors must be accessed by explicit assignments to local variables. Hence, an ICurry *case* statement can be directly mapped into a typical *switch* statement available in many imperative languages. To do this easily, the constructors $c_1, \ldots, c_n$ in the branches of a *case* statement are all the constructors of the data definition and in the same order.

[3] https://www-ps.informatik.uni-kiel.de/~cpm/pkgs/flatcurry.html

Local variable declarations and assignments are used to access arguments of a function call as well as the constructor arguments in case expressions. The values of local variables can be used to construct a new expression to be returned as the result of a function call. Such an expression is either a *variable*, an *application* of a constructor or function symbol to argument expressions, or a *disjunction*. In order to support various search strategies, non-determinism is not implemented as control code (like backtracking in Prolog), but explicitly represented in the graph structure (see below for more details). A variable in an expression is either the value of a local variable, the root of the function call (stored in a specific register *ROOT* when the function code is invoked), or $v[i]$, the i-th successor of a node identified by v.

For instance, a textual representation of the ICurry code for the operation `xor`, defined in Sect. 2, could be as follows (arguments are indexed starting with `0`):

```
function xor
  declare x1
  declare x2
  x1 = ROOT[0]
  x2 = ROOT[1]
  case x1 of
    False  → return x2
    True   → return NODE(not, x2)
```

In order to translate a FlatCurry program into an ICurry program, branches in *case* expressions must be ordered and completed, nested *case* and *let* constructs are eliminated by introducing new operations, etc. (see [15] for details). For instance, the translation of the operation `insert`, defined in Sect. 2, requires the introduction of a new auxiliary operation:

```
function insert
  declare x1
  declare x2
  x1 = ROOT[0]
  x2 = ROOT[1]
  return NODE(:, x1, x2) or NODE(insert_CASE, x2, x1)

function insert_CASE
  declare x2
  declare x1
  x2 = ROOT[0]
  x1 = ROOT[1]
  case x2 of
    []  → exempt
    :   → declare x3
          declare x4
          x3 = x2[0]
          x4 = x2[1]
          return NODE(:, x3, NODE(insert, x1, x4))
```

Note that an *exempt* statement has been introduced in the completion of the *case* statement in order to indicate the failure on empty lists of the operation `insert_CASE`.

A representation of ICurry programs together with a compiler from FlatCurry to ICurry is available as a Curry package.[4] ICurry has been used for various approaches to compile Curry into imperative target languages, like C, Python, Java, JavaScript, or native code [15, 18, 52]. Although the meaning of most constructs is obvious, the complete operational semantics contains some freedom w.r.t. the implementation of non-determinism. Since this is one of the main aspects of our work, we will fix the concrete evaluation strategy in the next two sections.

[4] https://www-ps.informatik.uni-kiel.de/~cpm/pkgs/icurry.html

5 PULL-TABBING

As mentioned in the introduction, our implementation works on a graph structure where non-deterministic choices are represented as graph nodes. This allows to use various, in particular, fair search strategies [17] to evaluate a program. Since a disjunction (*or* node) is not a defined operation, its evaluation must be fixed by the run-time system. We discuss this in the following.

In principle, there are various options to deal with non-deterministic choices ("don't know non-determinism") in declarative programming languages:

- *Backtracking* implements a choice by selecting one alternative and proceeding. If a computation is finished (with failure or success) and the next alternative should be explored, the state before a choice is restored and a next untried alternative is taken. This is used in Prolog implementations and also in implementations of functional logic languages that compile into Prolog, like PAKCS [7, 29] or TOY [40], or use Prolog-like implementation techniques [41]. Due to a possible non-termination of an alternative, some values might not be computed.
- *Copying* or *cloning* evaluates alternatives in parallel or by interleaving steps on different copies of the computation space that are made when a choice occurs. This ensures completeness (if the interleaving is fair) but can be costly when a choice occurs deeply in an expression. Thus, it has been used only in experimental implementations, e.g., [16].
- *Pull-tabbing* avoids the cost of cloning by keeping all alternatives in one graph structure and duplicating nodes on demand. It was first sketched in [3] and formally explored in [5]. A pull-tab step moves a choice in a demanded argument outside by copying the node representing the operation. For instance,

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

  is a pull-tab step. Since evaluation steps are performed in place in a graph-based implementation [43, 45], such a step sets the tag of the initial "`not`" node to the new tag "`?`" and creates two new nodes tagged with "`not`." Since pull-tabbing avoids the disadvantage of cloning and does not require a fixed backtracking strategy, it is used in implementations targeting complete search strategies, e.g., KiCS [20], KiCS2 [19], or Sprite [18]. Therefore, we will also use pull-tabbing but in a considerably improved form.

A problem of pure pull-tabbing is its unsoundness w.r.t. the call-time choice semantics sketched in Sect. 2. Consider the definition of `xorSelf` in Sect. 2. The expression `xorSelf (False ? True)` will be evaluated to

```
(False ? True) ? (True ? False)
```

by pure pull-tabbing. According to the call-time choice semantics, `True` is not a valid result. To avoid the extraction of such unintended values, each choice has a specific identifier [5]. When a choice is introduced in an expression, a fresh identifier is assigned to this choice. Pull-tab steps keep the identifier of the choice. Hence, the expression `xorSelf (False ?1 True)` will be evaluated to

```
(False ?1 True) ?1 (True ?1 False)
```

so that all choices have the same identifier. Furthermore, each task evaluating some branch of a computation has a *fingerprint* [18], i.e., a (partial) mapping from choice identifiers to choice alternatives (*L*eft or *R*ight). When a task evaluates a choice at the root, it proceeds as follows:

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

With this extension, the value `True` will not be extracted from the choice structure above.

By scheduling the tasks in various orders, pull-tabbing supports the application of various search strategies. Nevertheless, it has a serious drawback. As shown above, a single choice might result in multiple copies of this choice that need to be explored by tasks with their fingerprints. This could lead to a considerable overhead whenever a non-deterministic expression is shared multiple times during a computation, as discussed in [25]. A solution to this problem, recently sketched in [31], is the addition of a kind of memoization to pull-tabbing. In the next section, we formalize this strategy that is the basis of our implementation.

6 OPERATIONAL SEMANTICS

As already mentioned, the execution model of Curry is based on graph rewriting, i.e., a program is considered as a graph rewriting system [22, 45]: a rewrite step applies a rule to some subgraph by replacing the subgraph with the instantiated right-hand side of the rule. In the following, we extend a graph-based model for ICurry computations [15] with a refinement of pull-tabbing sketched in [31].

We review some notions of graph rewriting and fix our notation. We write $f(x) = \bot$ if the (partial) function is undefined on x. $\overline{o_k}$ denotes a sequence of objects $o_1, \ldots, o_k$. $\overline{o_k}[o]_j$ denotes the sequence $o_1, \ldots, o_{j-1}, o, o_{j+1}, \ldots, o_k$ where the object at position j is replaced by o.

A *graph* G is a set of *nodes*, where each node has a *label* (a constructor or function symbol) and a sequence of *successors*. Later we will add a few more attributes to nodes. We write $G[n] = s(\overline{n_k})$ if n is a node of G with label s and k successor nodes $\overline{n_k}$. Graph nodes correspond to expressions so that we use them interchangeably. A graph might also contain *choice nodes* of the form $?^c(n_1, n_2)$. c is a *choice identifier*, e.g., an integer or another element from an infinite set of constants [19], and the expressions n_1, n_2 are the alternatives.

We denote the *update* of a node n of G by $G[n \leftarrow s(\overline{n_k})]$. This replaces the label of n by s and the successors by $\overline{n_k}$. In order to implement sharing, it is sometimes necessary to redirect a graph node n to a node n' of a graph G, e.g., when a collapsing rule like "`id x = x`" is applied. We denote the *redirection* of node n to n' by $G[n \leftarrow n']$. This can be implemented either by a specific "redirection node" or by redirecting all edges pointing to n so that they point to n'. Finally, we denote the *extension* of a graph G with a new node n by $G \uplus \{n : s(\overline{n_k})\}$. The node n, which does not exist in G, has label s and k successors $\overline{n_k}$.

To deal with non-deterministic computations in a flexible manner, the run-time system works with a set or queue of computation tasks, where each task evaluates one expression. A *task* is a tuple (C, S, F, t) with:

- C is the *control* which is either a graph node n to be evaluated or a pair (b, E) consisting of an ICurry block (see Fig. 2) and an *environment* E (a mapping from local variables to graph nodes).
- S is a stack where each stack element is a node n labeled by a function symbol. The stack contains the functions to be evaluated by a task.
- F is a *fingerprint*: a (partial) mapping from choice identifiers to $\{1, 2\}$ (the indexes of alternatives).
- t is a *task identifier* to uniquely identify a task, e.g., a number or another constant taken from a conceptually infinite set. Task identifiers are used to store task-specific information in graph nodes.

Thus, the *state of a computation* is a triple (G, T, R) with:

- G is a graph.
- T is a set of tasks.
- R is a set of graph nodes representing the computed results.[5]

In the following, we use $\phi[x \mapsto v]$ to denote an update of a mapping ϕ for some argument v. If $\phi' = \phi[x \mapsto v]$, then $\phi'(x) = v$ and $\phi'(y) = \phi(y)$ for all $y \neq x$. Furthermore, we use Curry's list notation in states, e.g., we denote the set of tasks as a list (options to implement the set of tasks will be discussed later). Thus, an *initial state* of a computation has the form

$$(G, [(n, [], \{\}, t_0)], \{\})$$

where the graph G contains the initial expression with root node n and t_0 is the identifier of the initial task. An initial state contains only one task with an empty stack and fingerprint and an empty set of results.

A *final state* of a computation has the form:

$$(G, [], R)$$

There are no tasks left and the set R contains the root nodes of all computed results.

The operational semantics is specified by a set of transformation rules on computation states. We will use an auxiliary operation *extend* to extend a graph with the graph representation of an expression occurring in an ICurry program. Informally, $extend(G, E, e)$ extends a graph G with an ICurry expression e w.r.t. an environment E and returns the pair (G', n) consisting of the extended graph and the root node n of the added expression. To define *extend*, we use an auxiliary function *lookup* to retrieve a graph node w.r.t. an

[5] We specify the evaluation to head normal forms, i.e., graphs with a constructor at the root. This is sufficient since the evaluation to normal form can be implemented by auxiliary operations.

environment:

$$lookup(G, E, v) = \begin{cases} E(v) & \text{if } v = x \text{ or } v = ROOT \\ n_i & \text{if } v = v'[i],\ lookup(G, E, v') = n \\ & \text{and } G[n] = l(n_1, \ldots, n_k) \end{cases}$$

Now we can define $extend(G, E, e)$ by a case distinction on the ICurry expression e. If e is a variable, the graph is not extended and the binding of the variable is looked up in the environment E:

$$extend(G, E, v) = (G, lookup(G, E, v))$$

If e is a disjunction e_1 *or* e_2, new subgraphs for e_1 and e_2 are created and connected by a new choice node:

$$extend(G, E, e_1 \text{ or } e_2) = G'' \uplus \{n : ?^c(n_1, n_2)\}$$
$$\text{if } extend(G, E, e_1) = (G', n_1) \text{ and } extend(G', E, e_2) = (G'', n_2)$$

Here, c is a new choice identifier. We assume the existence of a global set of choice identifiers so that new unique choice identifiers can be obtained during the computation [19].

If e is a node constructor, new subgraphs for the argument expressions are created together with a new node connecting these subgraphs:

$$extend(G, E, NODE(l, \overline{e_k})) = G' \uplus \{n : l(\overline{n_k})\}$$

where n_i is the root node for the subgraph created for e_i ($i = 1, \ldots, k$) and G' is the graph containing G and the new subgraphs.

Each task evaluates an expression to head normal form, i.e., a constructor-rooted expression. It starts evaluating the function at the root by applying its defining rule. However, when the rule contains a *case* statement, the discriminating argument must be evaluated first. To do so, the current node is pushed onto the stack of the task and the discriminating node is evaluated. To simplify this process, we assume that *each ICurry function contains at most one case statement*, which can be obtained by replacing nested *case* statements by auxiliary operations (as also done in some implementations of Curry, e.g., [29]). Therefore, we denote by f^d an ICurry function which demands its d-th argument in a *case* statement, otherwise the superscript is omitted.[6]

The notion of graph rewriting is used to perform updates in place: if a node n containing a function call $f(\ldots)$ is reduced to some expression e, node n is replaced by e. This is important to avoid repeated evaluations of the same subexpression, e.g., the argument `(not True)` in the expression

```
xorSelf (not True)
```

Since the argument of `xorSelf` is shared (rather than duplicated), the update-in-place of the `not` node avoids a repeated evaluation. While this is the usual implementation of non-strict functional languages [43], the extension to non-deterministic languages is more involved. Consider the definition

```
and True  x = x
and False x = False
```

and the expression

```
let x = aBool in ... (and x x) ...
```

If some task evaluates the argument `x` to `False`, it would be wrong to replace `(and x x)` by `False`, since another task might evaluate `x` to `True` so that the replacement is wrong w.r.t. this task. This is the reason why pull-tabbing moves every choice to the top before some task selects one of the alternatives. As mentioned above, this could lead to a substantial overhead if a non-deterministic expression, like `x`, is shared and some task evaluates it several times: due to non-determinism, sharing is lost. For instance, consider a function

$$f\ x = C[x, \ldots, x]$$

where the argument x occurs n times in the right-hand side and every occurrence of x is demanded by the context C. If we evaluate $f\ (e_1\ ?\ e_2)$, pull-tabbing moves every demanded occurrence of the choice to the root so that approximately $n \cdot d$ pull-tab steps are performed if the n occurrences of x in C are at depth d. Since each pull-tab step duplicates nodes, a complex choice structure is introduced and finally processed to extract the two different values.

A solution is sketched as *memoized pull-tabbing* in [31]: function nodes are not updated in place but contain a store for task-specific updates. For this purpose, each graph node representing a function call contains a *task result map tr* which is a partial mapping from task identifiers to graph nodes. To formalize this idea, we denote by $tr_G[n]$ the task result map of node n in graph G, and by $tr_G(n)[t \mapsto n']$, where t is a task identifier and n' a node in graph G, the update of the task result map of node n so that $tr_G[n](t) = n'$.

Now we have all elements to specify the operational semantics of ICurry w.r.t. the memoized pull-tabbing strategy. The rules, summarized in Fig. 3, describe the steps to transform a non-final state of a computation. We explain these transformation rules in the following.

Control rules. The first block of rules organize the control to deal with choices and results and to invoke functions.

split choice: If the control contains a choice node and an empty stack, the choice is at the top. Since the fingerprint does not contain a selection for this choice, the choice is evaluated for the first time so that both alternatives need to be evaluated by independent tasks. Thus, the current task is replaced by two new tasks where the fingerprint of each task is extended accordingly. We assume the existence of a global set of task identifiers so that fresh ones can be obtained during the computation. In principle, one can use any strategy to add the new tasks to the existing ones. Here we put them at the end which corresponds to a *breadth-first strategy* in the search tree. Putting them at the front of T corresponds to a *depth-first search strategy*. One can also evaluate the tasks in parallel. Due to this flexibility, our implementation supports the selection of different search strategies.

select choice: If the control contains a choice node and an empty stack, i.e., the choice is at the top, and the fingerprint has already a selection for this choice, the choice node in the control is simply replaced by the selected branch.

pull tab: This rule describes a pull-tab step as discussed in Sect. 5. If the control contains a choice node and the stack is not empty, the choice is at a demanded argument position d of some function f referenced by the graph node n contained in the top of the stack. Then two new nodes are created by duplicating this function call

[6] In principle, one can extend the operational model of ICurry to more than one demanded position. However, our practical experiences showed a considerable runtime improvement when considering only one demanded operation since this allows to generate more efficient code. In general, operations with more than one demanded position are seldom so that it is difficult to obtain an operational advantage from considering them.

OVERALL CONTROL

(*split choice*)	$(G, (n, [], F, t) : T, R) \rightarrow$ $(G, T +\!\!+ [(n_1, [], F[c \mapsto 1], t_1), (n_2, [], F[c \mapsto 2], t_2)], R)$	if $G[n] = ?^c(n_1, n_2)$, $F(c) = \bot$ and t_1, t_2 are new task identifiers
(*select choice*)	$(G, (n, [], F, t) : T, R) \rightarrow (G, (n_j, [], F, t) : T, R)$	if $G[n] = ?^c(n_1, n_2)$ and $F(c) = j$ $(\neq \bot)$
(*pull tab*)	$(G, (n_0, n : S, F, t) : T, R) \rightarrow$ $(G'[n \leftarrow ?^c(n'_1, n'_2)], (n, S, F, t) : T, R)$	if $G[n_0] = ?^c(n_1, n_2)$, $F(c) = \bot$, $G[n] = f^d(\overline{n_k})$, $G' = G \uplus \{n'_1 : f^d(\overline{n_k}[n_1]_d), n'_2 : f^d(\overline{n_k}[n_2]_d)\}$
(*memo pull tab*)	$(G, (n_0, n : S, F, t) : T, R) \rightarrow$ $(G', (n, S, F, t) : T, R)$	if $G[n_0] = ?^c(n_1, n_2)$, $F(c) = j$, $G[n] = f^d(\overline{n_k})$, $G' = G \uplus \{n' : f^d(\overline{n_k}[n_j]_d)\}$, $tr_{G'}(n)[t \mapsto n']$
(*follow task result*)	$(G, (n, S, F, t) : T, R) \rightarrow (G, (n', S, F, t) : T, R)$	if $tr_G[n](t) = n'$ $(\neq \bot)$
(*case function*)	$(G, (n, S, F, t) : T, R) \rightarrow (G, (n_d, n : S, F, t) : T, R)$	if $tr_G[n](t) = \bot$ and $G[n] = f^d(\overline{n_k})$
(*non-case function*)	$(G, (n, S, F, t) : T, R) \rightarrow (G, ((b, \{ROOT \mapsto n\}), S, F, t) : T, R)$	if $tr_G[n](t) = \bot$, $G[n] = f(\cdots)$, and f defined by $f = b$
(*global result*)	$(G, (n, [], F, t) : T, R) \rightarrow (G, T, R \cup \{n\})$	if $G[n] = c(\cdots)$ for some constructor c
(*argument result*)	$(G, (n_0, (n : S, F, t) : T, R) \rightarrow$ $(G, ((b, \{ROOT \mapsto n'\}), S, F, t) : T, R)$	if $G[n_0] = c(\cdots)$ for some constructor c, $G[n] = f^d(\overline{n_k})$, f^d defined by $f^d = b$, $G' = G \uplus \{n' : f^d(\overline{n_k}[n_0]_d)\}$, $tr_{G'}(n)[t \mapsto n']$

EVALUATION OF BLOCKS

(*variable declaration*)	$(G, ((declare\ x; b, E), S, F, t) : T, R) \rightarrow (G, ((b, E[x \mapsto null]), S, F, t) : T, R)$	
(*local assign*)	$(G, ((x = e; b, E), S, F, t) : T, R) \rightarrow$ $(G', ((b, E[x \mapsto n]), S, F, t) : T, R)$	if $extend(G, E, e) = (G', n)$
(*successor assign*)	$(G, ((v[i] = e; b, E), S, F, t) : T, R) \rightarrow$ $(G'[n \leftarrow l(\overline{n_i}[n']_i)], ((b, E), S, F, t) : T, R)$	if $lookup(G, E, v) = n$, $G[n] = l(\overline{n_k})$, $extend(G, E, e) = (G', n')$
(*return*)	$(G, ((return\ e, E), S, F, t) : T, R) \rightarrow$ $(G'[E(ROOT) \leftarrow n], (n, S, F, t) : T, R)$	if $extend(G, E, e) = (G', n)$
(*exempt*)	$(G, ((exempt, E), S, F, t) : T, R) \rightarrow (G, T, R)$	
(*case*)	$(G, ((case\ x\ of\ \{c_1 \rightarrow b_1; \ldots; c_n \rightarrow b_n\}, E), S, F, t) : T, R) \rightarrow$ $(G, ((b_i, E) : S, F, t) : T, R)$	if $E(x) = n$ and $G[n] = c_i(\cdots)$

Figure 3: Operational semantics of ICurry with memoized pull-tabbing

but replacing the d-th argument by the two alternatives of the choice. The actual node at the top of the stack is replaced by a choice (with the same identifier) containing these two nodes and becomes the new control.

memo pull tab: This rule is an important part of the improved memoized pull-tabbing strategy. If the control contains a choice node at a demanded argument position and the actual task has a selection for this choice in its fingerprint, the choice is not moved to the top as in pure pull-tabbing. Instead, the selected alternative is put into the demanded argument. Since, due to sharing, it would be wrong to update the actual function node, a new function node is created with this update and the task result map of this original node refers to the new node for the current task identifier. Thanks to this rule, only the first occurrence of a choice is moved to the root by pull-tab steps.

follow task result: This rule is responsible to follow the correct alternative for shared non-deterministic expressions in a computation. If the task result map for a node contains an entry for the current task, the node referred by this entry becomes the new control element.

case function: If the control contains a graph node labeled with a defined function but no entry in its task result map, and its d-th argument is demanded, the function node is put onto the stack and the control is replaced by the d-th argument.

non-case function: If the control contains a graph node labeled with a defined function but no entry in its task result map and no demanded argument, the function is ready for evaluation. Hence, its body is put into the control together with an environment initialized with the specific variable *ROOT* set to the graph node with this function.

global result: If the control contains a graph node labeled with a constructor symbol and the stack is empty, a constructor-rooted term (head normal form) has been computed. This is added to the set of computed results and the current task is removed.

argument result: If the control contains a graph node labeled with a constructor symbol and the stack is not empty, then the top of the stack contains a function where the argument at position d is demanded and now evaluated. Since this result might be valid only for the current task, a new function node with this argument is introduced and remembered in the task result map (this could be relevant if the function is shared in the subsequent computation). After this preparation, the function is invoked similarly to rule (*non-case function*).

Evaluation of blocks. The remaining rules are responsible to evaluate the body of a single function contained in the control. It should be noted that these rules describe elementary computational activities so that they can be directly mapped into imperative programs.

variable declaration: If the control starts with a declaration of a local variable, it is initialized as a *null* pointer in the environment. The compilation scheme from FlatCurry into ICurry ensures that each variable will be assigned before its first use [15] so that *null* can be replaced by any other value.

local assign: If the control starts with an assignment to a local variable, the graph is extended with the expression and the environment is updated for this variable.

successor assign: If the control starts with an assignment to a successor of a node (such assignments might occur in the case of cyclic structures), the graph is extended with the expression and the successor is set to the created subgraph.

return: If the control contains a return statement, the graph is extended with the returned graph and the root node containing the current function is updated with the returned node.

exempt: If the control contains an exempt statement, the current computation is removed from the list of tasks.

case: If the control contains a case statement, the corresponding branch is selected. This selection is always possible since the case argument is demanded so that it has been evaluated before invoking the function and all case branches are complete.

Discussion. This operational model avoids repeated pull-tab steps for shared non-deterministic expressions by the use of task result maps for function nodes. The first occurrence of a non-deterministic choice is moved to the root by repeated applications of rule (*pull tab*). Then the computation is split into two tasks by rule (*split choice*). Each of these new tasks directly selects any further occurrence of the same choice to the appropriate branch by rule (*memo pull tab*). Since the further computation with this choice is valid only for the current task, the computed results are remembered in the task result maps.

The use of task result maps causes some overhead for deterministic computations, i.e., computations where choices do not occur. This overhead can be avoided by attaching task identifiers in nodes in order to identify the task which created a node. This information can be used in rule (*argument result*) to create the new node n' and update the task result map of n only if the computed result comes from a task "younger" than the current task. Since there is only one task for deterministic computations, the updates of the task result map are avoided.

This improvement is sketched in [31]. Since a detailed specification would be quite technical, we omit it in this formal model.

7 COMPILATION TO GO

After fixing the operational semantics of the intended implementation, we are ready to compile Curry programs into an existing target language. Since the transformation of Curry programs into ICurry programs via FlatCurry already exists, as specified in [15], we focus on the compilation of ICurry programs. We recapitulate the main issues of such an implementation:

(1) Implement the graph structure containing all expressions and choices under evaluation.
(2) Implement the overall control, i.e., the management of tasks.
(3) Implement the evaluation of single functions, i.e., pattern matching via case distinctions, assignments, and construction of new graphs.

In order to do this with limited efforts (which was one of the motivation of designing ICurry), an imperative target language supporting dynamic data structures and garbage collection is an appropriate choice. Moreover, to support fair evaluation of non-determinism and exploiting multi-processor architectures for running Curry programs, the target language should also support lightweight threads.

Due to these considerations, we choose Go[7] as the target language of our implementation. Go is a statically typed language with garbage collection and direct support for CSP-like concurrency [35]. New threads can be started as procedure calls prefixed by the keyword "go" so that they are also called *goroutines*. They run in the same address space (shared memory). Go also offers *channels* to communicate between goroutines. We will use channels to send computed results from the tasks to the main program presenting the results to the user. Since goroutines are multiplexed onto operating-system threads, multi-core processors are automatically exploited (if not explicitly restricted by some compiler option).

In the following, we sketch our implementation, called Curry2Go, with some simplifications to keep it readable. The complete implementation is available as a Curry package[8] and contains a compiler and an interactive environment ("REPL") in the style of Haskell, Prolog, or other Curry systems. Since the compiler and REPL are written in Curry, the initial installation requires another existing Curry compiler. Subsequently, it can be compiled with itself (bootstrapping).

7.1 Run-Time Structures

The main data structure of our run-time system is the graph representing expressions and choices occurring during the evaluation of the main expression. For this purpose, we have to define a structure for graph nodes. The various concrete kinds of nodes are distinguished by the following enumeration type in Go:

```
type NodeType uint8
const(
    FCALL NodeType = iota
```

[7] https://golang.org/
[8] https://github.com/curry-language/curry2go

```
    CONSTRUCTOR
    CHOICE
    REDIRECT
    EXEMPT
    ...
)
```

Hence, a node could be a function call (`FCALL`), a constructor application (`CONSTRUCTOR`), a choice (`CHOICE`), a redirection node (`REDIRECT`), or a failure (`EXEMPT`) (further alternatives are different kinds of literals, which are omitted here). Note that the operational semantics described in Sect. 6 has no failure nodes but specifies ICurry's *exempt* statement by dropping the current task. Since Curry2Go uses a separate layer for managing tasks and sending computed results, we implement *exempt* by creating an `EXEMPT` node which is processed by the task manager.

We use the following Go structure to implement graph nodes:

```
type Node struct{
    Children []*Node
    node_type NodeType
    int_value int
    function func(*Task)
    arity int
    name *string
    lock sync.Mutex
    tr map[int]*Node
}
```

Each node has an array `Children`, the successors of the node, i.e., pointers to other nodes, and a tag `node_type`. The use of the further components depends on the kind of node. For constructor nodes, `int_value` contains the index of the constructor which is used for the efficient implementation of case selection according to rule (*case*). For choice nodes, `int_value` contains the identifier of the choice (see Sect. 5). For function nodes, `int_value` contains the index of the demanded argument (or `-1` when the body of the function does not contain a case statement), and `function` refers to the Go function invoked to evaluate the function node (the reference to the task is necessary since it is relevant which task evaluates a function). In the case of function or constructor nodes, `arity` contains the number of required arguments: if the number of successors is smaller than the arity, the node represents a partial application. `name` refers to the textual representation of the constructor or function and is used for debugging or printing results. `tr` is the task result map used to implement memoized pull-tabbing as discussed in Sect. 6. To avoid the construction of a map structure for all graph nodes, this component will be initialized on demand, i.e., a new map structure is created when it is set for some value for the first time. Finally, `lock` is a mutual exclusion lock which is necessary to synchronize concurrent tasks evaluating and updating the graph (see below).

The representation of tasks in our run-time system has a direct correspondence to the operational semantics specified in Sect. 6:

```
type Task struct{
    id int
    control *Node
    stack []*Node
    fingerprint map[int]int
}
```

`id` is the task identifier (used in the task result maps), `control` refers to the node in the control, `stack` refers to the nodes in the stack of the task, and `fingerprint` maps choice identifiers into the indexes of children of choice nodes.

The ICurry block and environment, mentioned in the operational semantics, are not part of the task structure since they are represented as Go code which implements the rules to evaluate blocks as specified in Fig. 3. The structure and generation of this code is described below in Sect. 7.3.

7.2 Search Control

The overall control (i.e., the operation implementing the evaluation of an expression) uses a Go channel `result_chan` to receive evaluated results from individual tasks (this corresponds to the component R of the computation state of the operational semantics) and a queue of tasks. It is based on a function `toHNF` which evaluates the control node of a task to a head normal form or a failure. This is done by repeatedly invoking the code of defined operations. If a choice occurs and the fingerprint of the task has no selection for the choice, a new task is created by copying the current task and extending the fingerprints of the current and new task for this choice. In the case of a depth-first search (DFS) strategy, the new task is put into the front of the queue of tasks and we proceed with the current task. In the case of a breadth-first search (BFS) strategy, the current task and the new task is put at the end of the queue of tasks and we proceed with the first task in the queue. If the evaluation of a task terminates, the result is ignored in the case of an `EXEMPT` node or send to `result_chan` so that the overall control can print it. If the queue of tasks is not empty, the next task is taken from the queue and executed.

So far we have described a sequential search strategy where the user can choose between DFS and BFS. However, a main motivation of this work is to provide an operationally complete implementation. For instance, consider the following contrived example:

```
idND :: a → a
idND n = loop ? n ? loop
```

Semantically, `idND` is the identity function but, operationally, it is non-deterministically defined with looping alternatives. Although `0` is a value of `idND 0`, both DFS and BFS do not return any value but loop. To avoid such kind of incompleteness, we implemented a *fair search* (FS) strategy. FS evaluates each task concurrently as a goroutine: the Go function `fsRunner(t,result_chan)`, which evaluates a task `t` by `toHNF` and sends a non-failure result to `result_chan`, is started by

```
go fsRunner(t,result_chan)
```

This is done by the task handler implementing FS. After starting a new runner, the task handler waits for new tasks put by the runners into the queue of tasks (which is also implemented as a Go channel) and starts them (there is also an option to set a maximum for concurrent tasks). Thus, each non-deterministic branch is evaluated as a separate goroutine.

Note that all these goroutines work on the same global graph structure. Although they usually evaluate different parts of this graph, due to sharing it is possible that different goroutines evaluate the same node and update them in place, e.g., replacing a function

node by a constructor or by a choice node in a pull-tab step. In order to avoid inconsistencies due to concurrent updates, graph nodes have to be locked. This is the purpose of the component `lock` of a graph node. If some node has to be evaluated, it will be locked by the task so that at most one thread accesses the node or its task result map. If it is a choice node, the parent node must be locked since a possible pull-tab step will update the parent.

7.3 Code Generation

The implementation of the run-time structures and control routines described so far form the run-time system of our implementation. All components of the run-time system are bundled in a Go package named `gocurry`. Our compiler generates for each Curry module a corresponding Go package which imports `gocurry` and further compiled Curry modules.

As already mentioned, a concrete Curry module is transformed via FlatCurry into an ICurry program so that the actual compiler transforms ICurry programs into Go programs. For this purpose, each ICurry function is compiled into a Go function which has a single parameter: the task evaluating this function. The task parameter is used to initialize a local variable `root` with a reference to the control node. The values of all further local variables are accessed via `root`. Then the translation of an ICurry block into Go statements is straightforward: *case* is translated into a `switch` instruction, *NODE*, *or*, *exempt* into constructors for the corresponding graph nodes, and a *return* sets the `root` node before returning from the Go function. Furthermore, expressions are translated into statements which create new graph nodes. In the following, we sketch this compilation process.

In order to simplify the Go code to create graph nodes, the Curry2Go compiler generates a Go function for each kind of node occurring in an ICurry expression. For this purpose, the run-time system contains a generic function which sets a given node (first argument) to a constructor node where the index, arity, name, and children of the constructor are passed as further arguments:

```
func ConstCreate(root *Node,
                 constructor int, arity int,
                 name *string,
                 args ...*Node) *Node {
    root.Children = root.Children[:0]
    root.node_type = CONSTRUCTOR
    root.int_value = constructor
    root.arity = arity
    root.name = name
    root.Children = append(root.Children, args...)
    return root
}
```

Similarly, there are further functions in the run-time system to set nodes to function nodes, choice nodes, etc.

Each data type is translated into an array containing the textual representation of each constructor and operations to create constructor nodes. For instance, for the data type of Boolean values (defined in the `Prelude` module), the following Go code is generated:

```
var Prelude_Bool_names []string =
        []string{ "False", "True" }

func Prelude_CREATE_False(root *Node) *Node {
    ConstCreate(root, 0, 0, &Prelude_Bool_names[0])
    return root
}

func Prelude_CREATE_True(root *Node) *Node {
    ConstCreate(root, 1, 0, &Prelude_Bool_names[1])
    return root
}
```

The use of a global string array T`_names` for the textual representation of the constructors of a type T reduces memory and time compared to the direct storage of strings in constructor nodes. Similarly, the names of all operations occurring in some module are stored in a string array.

Note that different constructors of a data type can be distinguished by the component `int_value` in the constructor nodes (in the previous example: `0` for `False` and `1` for `True`). This enables the straightforward translation of *case* expressions into Go `switch` instructions. For instance, the Curry operation `not`, defined in the `Prelude` as in Sect. 2, is transformed via FlatCurry into the following ICurry code:

```
function not
  declare x1
  x1 = ROOT[0]
  case x1 of
    False  → return NODE(True)
    True   → return NODE(False)
```

The translation of this definition into Go code is basically a `switch` instruction where in each branch the node representing the function call is replaced by `True` and `False`, respectively.

```
func Prelude_not(task *Task) () {
    root := task.control // get root of function call
    var x1 *Node
    x1 = root.Children[0]
    switch x1.int_value {
        case 0:
            Prelude_CREATE_True(root)
            return
        case 1:
            Prelude_CREATE_False(root)
            return
    }
}
```

The Go code shown above has been simplified for the sake of readability, e.g., package qualifiers are omitted. Nevertheless, it shows that ICurry is an appropriate abstraction to generate imperative code from Curry programs, which was the main motivation to introduce ICurry, as discussed in [15].

7.4 Extensions

After sketching the main aspects of Curry2Go, we discuss some extensions necessary to support a complete Curry implementation.

Since Curry is intended to amalgamate functional and logic programming features, the usual higher-order constructs of functional programming must be supported. This is already prepared during

the transformation from Curry into FlatCurry: lambda abstractions are transformed into top-level functions with new names, and each application of an expression e_1 of functional type to some argument e_2 (if this is not a full or partial application of a defined operation or constructor) is transformed into the expression `apply(`e_1`,`e_2`)` where `apply` is a predefined operation. This technique is known as "defunctionalization" [47] and also used in logic programming to support higher-order features [50]. Therefore, Curry2Go interprets partially applied function calls as non-evaluable (which is the purpose of the component `arity` in graph nodes) and the operation `apply` is implemented by adding an argument and, if all arguments are provided, calls the actual function with all arguments.

So far we ignored free variables since, from a declarative point of view, they can be considered as non-deterministic operations generating their values [9]. For instance, the Curry expression

```
let x free in (x, not x)
```

can be replaced by

```
let x = aBool in (x, not x)
```

where the non-deterministic value generator `aBool` is defined as in Sect. 2.

Although such a transformation could be used to compile Curry into a deterministic language if a mechanism to interpret non-deterministic computation is provided, as done in KiCS2 with Haskell as a target languge [19], this approach has some drawbacks. If the possible set of values for free variables is infinite, computations with value generators might have larger or even infinite search spaces compared to Prolog, where free variables are treated as run-time objects and unification is used to solve term equations. In particular, unification can bind variables without instantiating them. Thus, unification can be considered as an optimized equality predicate where constraints between variables are stored instead of concrete variable bindings [14]. In order to get the same effect for Curry2Go, we have to represent free variables and their binding constraints at run time. This can be done by extending FlatCurry and ICurry with free variables (which is actually the case in the corresponding packages) and representing free variables as specific graph nodes. We implement a free variable as a constructor without successors and `int_value` "`-1`". This allows for an immediate implementation of narrowing, i.e., instantiating free variable arguments [46]. If an argument of some function is demanded, there is a `switch` instruction in the function's body. This `switch` instruction is extended with a case for `-1`: since this branch is selected if the demanded argument is a free variable, this branch binds the free variable to a graph representing a choice of the various constructors. Since this binding is valid only for the current task, the free variable node is not updated in place but the binding is stored in the task result map of this node (this has some similarity to binding arrays used in parallel implementations of Prolog [51] but the structure is different due to our requirements). With this implementation, we obtain the advantages of using free variables as in Prolog implementations while still allowing flexible and complete search strategies.

8 EVALUATION

The compiler from Curry to Go is written in Curry and based on existing packages to represent FlatCurry and ICurry in Curry and generating ICurry programs from Curry source programs. Thanks to this infrastructure, the size of the actual compiler is less than 400 lines of Curry code. The implementation of the run-time system in Go is also reasonable: the kernel (without auxiliary operations to implement predefined operations) consists of less than 500 lines of Go code.

The main motivation for this work is to explore whether an operationally complete implementation of Curry (and, thus, logic programming) is possible in a way competitive to other existing implementations. Therefore, the fair search strategy is the default strategy in Curry2Go. Using Go as a target language is useful to get a compact and maintainable implementation but it might result in less efficient code compared to using C or LLVM. Nevertheless, we compare our current implementation to other implementations of Curry.

For the benchmarks, we choose two other major Curry implementations. PAKCS [29], which is part of Debian and Ubuntu Linux distributions, compiles to Prolog (SWI-Prolog 8.0) and is based on backtracking. KICS2 [19] compiles to Haskell (GHC 8.4) where non-determinism is implemented by pure pull-tabbing (without memoization) so that various search strategies, like depth-first and breadth-first (default), are offered. Both Curry systems have been used for larger applications and provide many libraries and packages for application programming. Sprite [18] is a Curry implementation which uses LLVM to generate efficient target code. It is also based on pull-tabbing and ICurry and offers a fair evaluation strategy (without concurrency by rotating the queue of tasks every so often). We could not include it in the benchmarks since a working implementation is not available at the time of writing.[9] The published figures in [18] indicate that Sprite behaves similarly to KiCS2 (although KiCS2 does not support fair evaluation).

All benchmarks[10] were executed on a Linux machine running Debian 10 with an Intel Core i7-7700K (4.2GHz) processor with eight cores. We measured the elapsed time in seconds (with the `time` command as the average of three runs) of invoking the executable generated by the Curry compiler. The first table shows the timings for purely functional programs:

Example	PAKCS	KiCS2	Curry2Go
`nrev_4096`	8.25	0.44	1.17
`takPeano_24_16_8`	54.16	0.32	5.17
`takInt_24_16_8`	41.19	0.40	5.96
`revHO_1M`	15.03	0.36	2.31
`primesHO_1000`	38.81	0.44	4.11
`queens_10`	203.29	0.59	15.81

`nrev_4096` is the quadratic naive reverse algorithm applied to a list with 4096 elements, `takPeano` and `takInt` is a highly recursive function on naturals [42] applied to arguments (24, 16, 8) in Peano representation or with built-in integers, respectively. The remaining benchmarks use higher-order functions: `revHO_1M` reverses a list with one million elements in linear time using `foldl` and `flip`,

[9] Personal communication with Andy Jost.
[10] The benchmark suite is contained in the implementation of Curry2Go.

Table 1: Non-deterministic programs with various search strategies

Example	PAKCS	KiCS2 (DFS)	KiCS2 (BFS)	C2Go (DFS)	C2Go (BFS)	C2Go (FS)
`psort_13`	16.43	0.77	2.89	5.34	5.43	7.79
`addNum_2`	0.19	0.99	1.78	0.44	0.43	0.36
`addNum_5`	0.22	3.22	5.15	1.03	1.03	0.52
`addNum_10`	0.28	10.04	15.49	2.44	2.43	0.69
`select_50`	0.14	0.62	0.69	0.11	0.09	0.09
`select_100`	0.45	4.95	5.23	0.14	0.15	0.10
`select_150`	1.06	21.31	26.11	0.23	0.24	0.12
`isort_primes4`	15.74	0.42	0.42	1.76	1.76	1.72
`psort_primes4`	156.28	0.40	0.41	1.71	1.74	0.95

`primesHO_1000` computes the 1000th prime number via the sieve of Eratosthenes, and `queens_10` computes the number of safe positions of 10 queens on a 10×10 chess board.

These benchmarks indicate that, for purely functional programs, Curry2Go is much faster than PAKCS but less efficient than KiCS2. This is not surprising since Haskell/GHC is highly optimized for these kinds of programs. Actually, a native implementation of the naive reverse algorithm in Go, Haskell, and Prolog shows that Haskell/GHC is more than two times faster than Go and SWI-Prolog is three times slower than Go. The reason for the even less efficient behavior of PAKCS on this benchmark is the fact that the implementation of lazy evaluation in Prolog causes some overhead.

However, the situation becomes different when non-determinism is used, as shown in Table 1. Whereas KiCS2 is quite efficient on `psort_13`, the naive permutation sort applied to a list of 13 elements, it changes when non-deterministic expressions are shared: `addNum_`n non-deterministically chooses a number (out of 2000) and adds it n times, and `select_`n non-deterministically selects an element in a list of length n and sums up the element and the list without the selected element. The duplication of choices, mentioned at the end of Sect. 5, causes a serious slowdown which is avoided in our implementation thanks to memoized pull-tabbing. It is also worth to compare the timings when different search strategies (DFS, BFS, FS) are used. Whereas KiCS2 shows some overhead of breadth-first compared to depth-first (possibly due to the additional structures used to implement a breadth-first tree traversal), the numbers are almost identical for Curry2Go since the difference between BFS and DFS is just a different schedule of tasks. Apart from its operational completeness, the fair search (FS) strategy is sometimes faster than BFS or DFS thanks to the use of goroutines possibly scheduled on different processors.

An interesting effect of our implementation is shown in the last two lines of Table 1 (inspired by [19]). It shows the time to sort

```
[primes!!303, primes!!302, primes!!301, primes!!300]
```

with the deterministic insertion sort (`isort`) and the non-deterministic permutation sort (`psort`) algorithm, respectively, where `primes` defines the infinite list of all prime numbers. Due to backtracking, identical computations might be repeated if they occur in different non-deterministic branches. Thus, `primes` is re-evaluated by PAKCS several times when the list is passed to the non-deterministic operation `psort`. This is not the case in implementations based on pull-tabbing: due to sharing common subexpressions in the global graph, results of deterministic computations are *shared across non-determinism* [20]. This property has the advantage that a user of some library operation must not care about the internal implementation of the operation, e.g., whether it exploits non-determinism. With our implementation, where non-deterministic branches are evaluated by goroutines, it could be even better to use a non-deterministic implementation since it might map evaluations of common subexpressions to different computation nodes, as shown by the results for `psort_primes4`.

As a final benchmark, we show the timings for `psort_primes4` increased to a list of eight prime numbers and executed with different numbers of processors (by setting the Go variable `GOMAXPROCS`):

# processors	1	2	4	8
`psort_primes8`	6.57	3.41	1.99	1.55

The benchmark shows that the presence of multiple processors is exploited in a non-deterministic program without requiring specific user annotations.

We have seen that KiCS2 generates efficient code for deterministic programs. This could be explained with the highly optimized Haskell compiler GHC used as a back end (KiCS2 invokes GHC with option `-O2`). Since good code generation requires time, we performed a larger test to compare the compilation times. Since both KiCS2 and Curry2Go are implemented in Curry, we measured the time to bootstrap both compilers where the first phase used the same Curry compiler (PAKCS) and the second phase used the compiler generated in the first phase. In this experiment, Curry2Go was more than three times faster than KiCS2. Although the absolute run times are difficult to compare (due to different compiler source files and techniques to compile base libraries), it shows that the generation of good native code requires time, but our compiler would benefit from future optimization options of the Go compiler.

9 RELATED WORK

Early implementations of functional logic languages were often based on Prolog due to the direct support of non-determinism and free variables [7, 40]. Even implementations which compile into other target languages use backtracking to implement non-determinism, since the efficient implementation of backtracking,

known from Prolog [1], can be transferred to other compilation schemes (e.g., [24, 41]) or simulated with functional programming techniques [48]. However, the use of backtracking results in the operational incompleteness of the implementation.

Later on, some approaches use concurrent programming features for the implementation of functional logic languages. For instance, an abstract machine for Curry is implemented in Java where choices are implemented as Java threads [30], but the overall efficiency is low compared to other sequential implementations. KiCS2 [19] offers a parallel search strategy based on a parallel tree search library for Haskell, but operational completeness is not ensured in the case of infinite deterministic computations.

A systematic approach to operationally complete implementations of functional logic languages is the fair scheme [17], which is also the basis of Curry compiler Sprite [18]. Since the fair scheme is based on pull-tabbing, it suffers from the copying of choices as discussed in Sect. 5. This motivated our work in order to show that the fair scheme can be improved and implemented in a way which can compete with other incomplete schemes.

Related to our work are also parallel implementations of Prolog [21]. Such implementations might support Or-parallelism [51] (apply all rules in parallel) or And-parallelism [34] (prove conjunctions of literals in parallel). Andorra Prolog [33] tries to combine both kinds of parallelism. The main motivation of these approaches is to speed up the execution of programs while keeping their sequential properties. This is in contrast to our work where we aim at an implementation which frees the programmer from considering the influence of the evaluation strategy to computed results.

10 CONCLUSIONS

Implementations of declarative programming languages often trade efficiency for completeness. Although equations, as used in functional programs, have a clear interpretation ("replace equals by equals until a value is obtained"), functional programmers have to consider the peculiarities of the concrete programming language (e.g., strict evaluation of arguments vs. non-strict evaluation of branches of conditions, order of pattern matching, default rules) to ensure that intended results are actually computed. Similarly, logic programs can be interpreted as logic formulas but, due to backtracking in Prolog, one has to consider the order of search in concrete programs. Such considerations are somehow contrary to the idea of declarative programming where one wants to abstract from execution details. Therefore, an operationally complete implementation can support a higher-level programming style and has also advantages for teaching declarative programming.

In this work we presented a new approach towards this objective. We developed an operationally complete implementation by using recent advances in operational models for declarative programming. By representing computational alternatives as data rather than program code, our implementation supports various search strategies, in particular, strategies with a fair selection of non-deterministic choices. By mapping alternative computations to lightweight threads, i.e., goroutines, we obtain reduced elapsed times on multi-core processors. We presented a technique to compile Curry source programs into Go target programs by a stepwise transformation via FlatCurry and ICurry. In order to avoid potential efficiency problems when demand-driven evaluation is combined with non-determinism, we used a recent method, memoized pull tabbing, and provided a formal definition of this strategy. Using this formal model together with an existing infrastructure for processing Curry programs, the actual implementation could be done with modest efforts. In principle, one can use any other imperative language as a target language (actually, compilation to C or LLVM might get more efficient target code), but the choice of Go limits the implementation efforts due to the direct support of dynamic data structures with garbage collection and concurrency with threads.

Although our implementation is a first prototype, it has already been used to compile larger programs, like program analyzers, browsers, and documentation tools, and it is able to bootstrap the compiler in order to get a self-hosting system.

For future work, we will look into possibilities to speed up the target code and add more computational features, like set functions to encapsulate search [10] or residuation to support concurrency in source programs [26].

ACKNOWLEDGMENTS

The authors are grateful to the anonymous referees for helpful comments and suggestions to improve the paper.

REFERENCES

[1] H. Aït-Kaci. 1991. *Warren's Abstract Machine.* MIT Press.
[2] E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. 2005. Operational Semantics for Declarative Multi-Paradigm Languages. *Journal of Symbolic Computation* 40, 1 (2005), 795–829.
[3] A. Alqaddoumi, S. Antoy, S. Fischer, and F. Reck. 2010. The Pull-Tab Transformation. In *Proc. of the Third International Workshop on Graph Computation Models*. Enschede, The Netherlands, 127–132. Available at http://gcm2010.imag.fr/pages/gcm2010-preproceedings.pdf.
[4] S. Antoy. 1997. Optimal Non-Deterministic Functional Logic Computations. In *Proc. International Conference on Algebraic and Logic Programming (ALP'97)*. Springer LNCS 1298, 16–30.
[5] S. Antoy. 2011. On the Correctness of Pull-Tabbing. *Theory and Practice of Logic Programming* 11, 4-5 (2011), 713–730. https://doi.org/10.1017/S1471068411000263
[6] S. Antoy, R. Echahed, and M. Hanus. 2000. A Needed Narrowing Strategy. *J. ACM* 47, 4 (2000), 776–822. https://doi.org/10.1145/347476.347484
[7] S. Antoy and M. Hanus. 2000. Compiling Multi-Paradigm Declarative Programs into Prolog. In *Proc. International Workshop on Frontiers of Combining Systems (FroCoS'2000)*. Springer LNCS 1794, 171–185.
[8] S. Antoy and M. Hanus. 2005. Declarative Programming with Function Patterns. In *Proceedings of the International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'05)*. Springer LNCS 3901, 6–22. https://doi.org/10.1007/11680093_2
[9] S. Antoy and M. Hanus. 2006. Overlapping Rules and Logic Variables in Functional Logic Programs. In *Proceedings of the 22nd International Conference on Logic Programming (ICLP 2006)*. Springer LNCS 4079, 87–101.
[10] S. Antoy and M. Hanus. 2009. Set Functions for Functional Logic Programming. In *Proceedings of the 11th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'09)*. ACM Press, 73–82. https://doi.org/10.1145/1599410.1599420
[11] S. Antoy and M. Hanus. 2010. Functional Logic Programming. *Commun. ACM* 53, 4 (2010), 74–85. https://doi.org/10.1145/1721654.1721675
[12] S. Antoy and M. Hanus. 2012. Contracts and Specifications for Functional Logic Programming. In *Proc. of the 14th International Symposium on Practical Aspects of Declarative Languages (PADL 2012)*. Springer LNCS 7149, 33–47. https://doi.org/10.1007/978-3-642-27694-1_4
[13] S. Antoy and M. Hanus. 2017. Default Rules for Curry. *Theory and Practice of Logic Programming* 17, 2 (2017), 121–147. https://doi.org/10.1017/S1471068416000168
[14] S. Antoy and M. Hanus. 2017. Transforming Boolean equalities into constraints. *Formal Aspects of Computing* 29, 3 (2017), 475–494. https://doi.org/10.1007/s00165-016-0399-6
[15] S. Antoy, M. Hanus, A. Jost, and S. Libby. 2020. ICurry. In *Declarative Programming and Knowledge Management - Conference on Declarative Programming (DECLARE 2019)*. Springer LNCS 12057, 286–307. https://doi.org/10.1007/978-3-030-46714-2_18

[16] S. Antoy, M. Hanus, J. Liu, and A. Tolmach. 2005. A Virtual Machine for Functional Logic Computations. In *Proc. of the 16th International Workshop on Implementation and Application of Functional Languages (IFL 2004)*. Springer LNCS 3474, 108–125.

[17] S. Antoy and A. Jost. 2013. Compiling a Functional Logic Language: The Fair Scheme. In *Proceedings of the 23rd International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2013)*. Springer LNCS 8901, 202–219. https://doi.org/10.1007/978-3-319-14125-1_12

[18] S. Antoy and A. Jost. 2016. A New Functional-Logic Compiler for Curry: Sprite. In *Proceedings of the 26th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2016)*. Springer LNCS 10184, 97–113. https://doi.org/10.1007/978-3-319-63139-4_6

[19] B. Braßel, M. Hanus, B. Peemöller, and F. Reck. 2011. KiCS2: A New Compiler from Curry to Haskell. In *Proc. of the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011)*. Springer LNCS 6816, 1–18. https://doi.org/10.1007/978-3-642-22531-4_1

[20] B. Braßel and F. Huch. 2007. On a Tighter Integration of Functional and Logic Programming. In *Proc. APLAS 2007*. Springer LNCS 4807, 122–138.

[21] J. Chassin de Kergommeaux and P. Codognet. 1994. Parallel Logic Programming Systems. *ACM Computing Serveys* 26, 3 (1994), 295–336. https://doi.org/10.1145/185403.185453

[22] R. Echahed and J.-C. Janodet. 1997. *On Constructor-based Graph Rewriting Systems*. Research Report IMAG 985-I. IMAG-LSR, CNRS, Grenoble.

[23] J.C. González-Moreno, M.T. Hortalá-González, F.J. López-Fraguas, and M. Rodríguez-Artalejo. 1999. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming* 40 (1999), 47–87.

[24] M. Hanus. 1990. Compiling Logic Programs with Equality. In *Proc. of the 2nd Int. Workshop on Programming Language Implementation and Logic Programming*. Springer LNCS 456, 387–401.

[25] M. Hanus. 2012. Improving Lazy Non-Deterministic Computations by Demand Analysis. In *Technical Communications of the 28th International Conference on Logic Programming*, Vol. 17. Leibniz International Proceedings in Informatics (LIPIcs), 130–143. https://doi.org/10.4230/LIPIcs.ICLP.2012.130

[26] M. Hanus. 2013. Functional Logic Programming: From Theory to Curry. In *Programming Logics - Essays in Memory of Harald Ganzinger*. Springer LNCS 7797, 123–168. https://doi.org/10.1007/978-3-642-37651-1_6

[27] M. Hanus. 2017. CurryCheck: Checking Properties of Curry Programs. In *Proceedings of the 26th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2016)*. Springer LNCS 10184, 222–239. https://doi.org/10.1007/978-3-319-63139-4_13

[28] M. Hanus. 2020. Combining Static and Dynamic Contract Checking for Curry. *Fundamenta Informaticae* 173, 4 (2020), 285–314. https://doi.org/10.3233/FI-2020-1925

[29] M. Hanus, S. Antoy, B. Braßel, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, F. Steiner, and F. Teegen. 2020. PAKCS: The Portland Aachen Kiel Curry System. Available at http://www.informatik.uni-kiel.de/~pakcs/.

[30] M. Hanus and R. Sadre. 1999. An Abstract Machine for Curry and its Concurrent Implementation in Java. *Journal of Functional and Logic Programming* 1999, 6 (1999).

[31] M. Hanus and F. Teegen. 2021. Memoized Pull-Tabbing for Functional Logic Programming. In *Proc. of the 28th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2020)*. Springer LNCS 12560, 57–73. https://doi.org/10.1007/978-3-030-75333-7_4

[32] M. Hanus (ed.). 2016. Curry: An Integrated Functional Logic Language (Vers. 0.9.0). Available at http://www.curry-lang.org.

[33] S. Haridi and P. Brand. 1988. Andorra Prolog: An Integration of Prolog and Committed Choice Languages. In *Proc. Int. Conf. on Fifth Generation Computer Systems*. 745–754.

[34] M.V. Hermenegildo and F. Rossi. 1995. Strict and Non-Strict Independent And-Parallelism in Logic Programs: Correctness, Efficiency, and Compile-time Conditions. *Journal of Logic Programming* 22, 1 (1995), 1–45.

[35] C.A.R. Hoare. 1978. Communicating sequential processes. *Commun. ACM* 21, 8 (1978), 666–677. https://doi.org/10.1145/359576.359585

[36] G. Huet and J.-J. Lévy. 1991. Computations in Orthogonal Rewriting Systems. In *Computational Logic: Essays in Honor of Alan Robinson*, J.-L. Lassez and G. Plotkin (Eds.). MIT Press, 395–443.

[37] H. Hussmann. 1992. Nondeterministic Algebraic Specifications and Nonconfluent Term Rewriting. *Journal of Logic Programming* 12 (1992), 237–255. https://doi.org/10.1016/0743-1066(92)90026-Y

[38] T. Johnsson. 1985. Lambda Lifting: Transforming Programs to Recursive Functions. In *Functional Programming Languages and Computer Architecture*. Springer LNCS 201, 190–203.

[39] J. Launchbury. 1993. A Natural Semantics for Lazy Evaluation. In *Proc. 20th ACM Symposium on Principles of Programming Languages (POPL'93)*. ACM Press, 144–154.

[40] F. López-Fraguas and J. Sánchez-Hernández. 1999. TOY: A Multiparadigm Declarative System. In *Proc. of RTA'99*. Springer LNCS 1631, 244–247.

[41] W. Lux. 1999. Implementing Encapsulated Search for a Lazy Functional Logic Language. In *Proc. 4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99)*. Springer LNCS 1722, 100–113.

[42] W. Partain. 1993. The nofib Benchmark Suite of Haskell Programs. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*. Springer, 195–202.

[43] S.L. Peyton Jones. 1992. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of Functional Programming* 2, 2 (1992), 127–202.

[44] S. Peyton Jones (Ed.). 2003. *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press.

[45] D. Plump. 1999. Term Graph Rewriting. In *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages and Tools*, H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg (Eds.). World Scientific, 3–61.

[46] U.S. Reddy. 1985. Narrowing as the Operational Semantics of Functional Languages. In *Proc. IEEE Internat. Symposium on Logic Programming*. Boston, 138–151.

[47] J.C. Reynolds. 1972. Definitional Interpreters for Higher-Order Programming Languages. In *Proceedings of the ACM Annual Conference*. ACM Press, 717–740.

[48] P. Wadler. 1985. How to Replace Failure by a List of Successes. In *Functional Programming and Computer Architecture*. Springer LNCS 201, 113–128.

[49] P. Wadler. 1997. How to Declare an Imperative. *Comput. Surveys* 29, 3 (1997), 240–263.

[50] D.H.D. Warren. 1982. Higher-order extensions to Prolog: are they needed?. In *Machine Intelligence 10*. 441–454.

[51] D.H.D. Warren. 1987. The SRI Model for Or-Parallel Execution of Prolog: Abstract Design and Implementation Issues. In *Proc. of the 1987 Internat. Symposium on Logic Programming*. IEEE-CS, 92–102.

[52] M.A. Wittorf. 2018. *Generic Translation of Curry Programs into Imperative Programs (in German)*. Master's thesis. Kiel University.