TPLP: Page 1–40.

doi:10.1017/S1471068426100453

*A Monadic Implementation of Functional Logic Programs**

MICHAEL HANUS, KAI-OLIVER PROTT and FINN TEEGEN
Department of Computer Science, Kiel University, Germany
(*e-mails:* mh@informatik.uni-kiel.de, kpr@informatik.uni-kiel.de, fte@informatik.uni-kiel.de)

submitted 4 March 2025 UTC; revised 1 May 2026 UTC; accepted 27 May 2026 UTC

Abstract

Functional logic languages are a high-level approach to programming by combining the most important declarative features. They abstract from small-step operational details so that programmers can concentrate on the logical aspects of an application. This is supported by appropriate evaluation strategies. Demand-driven evaluation from functional programming is amalgamated with non-determinism from logic programming so that solutions or values are computed whenever they exist. This frees the programmer from considering the influence of an operational strategy on the success of a computation, but it is a challenge to the language implementer. A non-deterministic demand-driven strategy might duplicate unevaluated choices of an expression, which could duplicate the computational effort. In recent implementations, this problem has been tackled by adding a kind of memoization of non-deterministic choices to the expression under evaluation. Since this has been implemented in imperative target languages, it was unclear whether this could also be supported in a functional programming environment like Haskell. This paper presents a solution to this challenge by transforming functional logic programs into a monadic representation. Although this transformation is not new, we present an implementation of the monadic interface which supports memoization in non-deterministic branches. Additionally, we include more advanced features of functional logic languages, namely functional patterns and encapsulated search, in our approach. By optimizing our implementation for purely functional computations with both a static and dynamic approach, we are able to achieve a promising performance that outperforms current compilers for Curry.

KEYWORDS: declarative programming, non-determinism, memoization, monads, implementation

1 Introduction

Declarative programming emphasizes the principle of expressing properties of a given problem in a high-level and execution-independent manner. In functional programming languages, equations specify the meaning of functions applied to given argument patterns.

* This is a revised and extended version of (Hanus *et al.* 2022), invited as a rapid communication in TPLP. The authors acknowledge the assistance of the conference program chairs Beniamino Accattoli and Manuel Hermenegildo.

These equations are used to reduce an initial expression to a value. In logic programming languages, the meaning of predicates or relations is specified by Horn formulas (implications). The non-deterministic resolution principle (Robinson 1965) uses these formulas to compute solutions to a given query.

Functional logic languages (Antoy and Hanus 2010; Hanus 2013) combine these programming paradigms in a single language environment. In order to abstract from small-step operational details, appropriate evaluation strategies are required. Lazy or demand-driven strategies ensure that iterated reduction steps w.r.t. given equations compute a value if it exists (Huet and Lévy 1991). Non-deterministic applications of program rules with overlapping left-hand sides ensure that solutions are computed whenever they exist (Lloyd 1987). The combination of these techniques is called *narrowing* (Slagle 1974; Reddy 1985). *Needed narrowing* is a demand-driven variant which is optimal w.r.t. the length of successful derivations and the number of computed solutions (Antoy 1997; Antoy *et al.* 2000).

However, the combination of demand-driven and non-deterministic evaluation steps might cause efficiency problems in concrete implementations. To sketch this problem, consider the following operations written in Curry (Hanus 2016):[1]

```
not False = True        aBool = False ? True                    Curry
not True  = False
```

`not` is a standard function whereas `aBool` is a *non-deterministic operation* (González-Moreno *et al.* 1999) which uses Curry's archetypal *choice* operation "?" that returns one of its arguments. Non-deterministic operations could have more than one value for a given input, for example `aBool` has values `False` and `True`. They are an important concept of contemporary functional logic languages (see (Antoy and Hanus 2010; González-Moreno *et al.* 1999) for more details). Non-deterministic operations can be used in data structures or as arguments to functions like any other operation, such as in "`not aBool`". Since the evaluation of any (sub)expression might lead to a non-deterministic choice, the occurrences of choices must be handled by the run-time system of the language implementation.

Some implementations use *backtracking* to handle non-determinism, in particular, implementations of functional logic languages that compile into Prolog, like PAKCS (Antoy and Hanus 2000; Hanus *et al.* 2025) or TOY (López-Fraguas and Sánchez-Hernández 1999). Backtracking implements a choice by selecting one alternative to proceed with the computation. If a computation terminates (with success or failure), the state before the choice is restored, and the next alternative is taken. A well-known disadvantage of backtracking is its operational incompleteness: if the first alternative does not terminate, no result will be computed. This problem can be avoided by keeping all alternatives in one computation structure (e.g., a graph of expressions) and using a fair strategy to explore this structure.

Pull-tabbing is an approach to implement this idea. It was first sketched in (Alqaddoumi *et al.* 2010) and formally explored in (Antoy 2011). A pull-tab step is a local transformation that moves a choice in a (demanded) argument of an operation outside this operation. For instance,

[1] Since the syntax of Curry is close to Haskell (Peyton Jones 2003) and we are going to compile Curry programs into Haskell programs, we denote the concrete language used in examples at the right margin.

https://doi.org/10.1017/S1471068426100453 Published online by Cambridge University Press

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

is a pull-tab step. Pull-tabbing is used in implementations targeting complete search strategies, for example KiCS (Braßel and Huch 2007), KiCS2 (Braßel *et al.* 2011), or Sprite (Antoy and Jost 2016). Iterated pull-tab steps move choices to the root of an expression. If expressions containing choices are shared (e.g., by applying rules with multiple occurrences of parameters in their right-hand sides), pull-tab steps might produce multiple copies of the same choice. This could lead to unsoundness and duplication of computations. The latter is a serious problem of pull-tabbing implementations (Hanus 2012). For instance, consider the additional operations

```
xor False x = x          xorSelf x = xor x x
xor True  x = not x
```
Curry

From a logical point of view, `xor` applied to identical Boolean values returns `False`. Thus, `xorSelf` should always return `False`. However, pull-tabbing transforms the single choice occurring in the expression `xorSelf aBool` into three choice occurrences.

```
xorSelf aBool  →  xor aBool aBool  →  ...
               →  (False ? True) ? (True ? False)
```

The value `True` (occurring twice) is incorrect. Note that this is caused by the combination of lazy (demand-driven) evaluation (which passes unevaluated expressions as arguments) and non-determinism, which is evaluated on demand. The problem of unsoundness can be fixed by attaching identifiers to choices (Antoy 2011). However, this does not fix the duplication of choices, which can be detrimental to performance due to repeated evaluation.

The Curry compiler KiCS2 (Braßel *et al.* 2011) transforms Curry programs into Haskell programs and uses pull-tabbing to handle non-determinism and to offer various search strategies. Actually, KiCS2 attaches identifiers to choices and, thus, suffers from the performance problem sketched above. Hanus (2012) proposed an optimization which eagerly evaluates demanded non-deterministic sub-expressions. This optimization requires a demand analysis which is non-trivial and often imprecise for complex data structures.

Recently, pull-tabbing has been improved by adding a kind of memoization so that the re-evaluation of shared non-deterministic choices is avoided. This scheme, called *memoized pull-tabbing* (MPT) (Hanus and Teegen 2021), has been used in Curry implementations which transform source programs into Julia programs (Hanus and Teegen 2021) or Go programs (Böhm *et al.* 2021).

Since MPT has been implemented only in imperative target languages, it is still open to debate whether the "MPT scheme can be combined with the purely functional implementation approach of KiCS2" (Hanus and Teegen 2021). A solution to this could be useful since KiCS2 supports maintainability by its high-level target language and is also highly efficient for purely functional computations (Braßel *et al.* 2011; Böhm *et al.* 2021).

In this paper, we present a solution to this challenge by transforming functional logic programs into a monadic representation. Although such a transformation is not new (Wadler 1990; Fischer *et al.* 2011), we present an implementation of the monadic interface which supports memoization of non-deterministic branches and also advanced features of

https://doi.org/10.1017/S1471068426100453 Published online by Cambridge University Press

contemporary functional logic languages, such as functional patterns (Antoy and Hanus 2005) and encapsulation of non-determinism (Braßel *et al.* 2004).

Our major contributions are:

1. A basic monadic model for Curry that uses memoization to avoid repeated computations.
2. Refinements of this basic model for efficiency improvements.
3. Extensions to cover advanced features of functional logic languages.

While we do not show the implementation of all extensions in detail in this paper, our implementation is the first Curry implementation that integrates all major features of Curry with a promising performance. A compiler that supports the full range of Curry features based on the presented approach is available at https://github.com/Ziharrk/kmcc.

This paper is structured as follows. The next section reviews some necessary details about functional logic programming. Section 3 presents the transformation of functional logic programs into purely functional programs parameterized by a monad to handle computational effects. Some relevant implementations of pull-tabbing are sketched in Section 4, followed by the implementation of our memoization monad in Section 5. This monad is extended in Section 6 to cover various features of contemporary functional logic languages, like free (logic) variables and unification, encapsulated search, and fair search strategies. In Section 7 we present optimization techniques to achieve a better performance for deterministic computations. We evaluate our approach in Section 8 before we discuss related work in Section 9 and conclude in Section 10.

2 Functional logic programming

We assume familiarity with basic ideas of logic and functional programming and the language Haskell, which we use for our implementation. In the following, we review only concepts of functional logic languages which are addressed by the implementation presented later. Concrete examples are shown in the multi-paradigm declarative language Curry.[2] More details can be found in surveys on functional logic programming (Antoy and Hanus 2010; Hanus 2013) and in the language report (Hanus 2016).

Functional logic languages amalgamate distinguishing 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 language Curry has a Haskell-like syntax[3] (Peyton Jones 2003) but allows *free* (*logic*) *variables* in conditions and right-hand sides of defining rules. In contrast to Haskell and similarly to logic programming, rule selection is non-deterministic, that is if more than one rule is applicable, all applicable rules are tried. The operational semantics is based on an optimal evaluation strategy (Antoy 1997; Antoy *et al.* 2000) – a conservative extension of lazy functional programming and logic programming.

[2] https://www.curry-lang.org

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

https://doi.org/10.1017/S1471068426100453 Published online by Cambridge University Press

As an example, consider the "classical" functional logic definition of the operation `last` to compute the last element of a list ("`++`" is the standard list concatenation operator, "`=:=`" denotes unification, and free variables are introduced by the keyword `free`):

```
last xs | ys ++ [x] =:= xs = x                                    Curry
  where x,ys free
```

This definition uses a conditional rule where the condition is solved by evaluating `ys ++ [x]` (which instantiates `ys` to some list) and unifying the result with the input list `xs`.

As mentioned in Section 1, *non-deterministic operations* (González-Moreno *et al.* 1999) are an important feature of contemporary functional logic languages. They are conceptually equivalent to free variables (as shown in (Antoy and Hanus 2006)) and can be nested like other functions. This is due to the fact that non-deterministic operations return (non-deterministically) individual values rather than sets of values. Their declarative meaning can be specified with a set-valued semantics (González-Moreno *et al.* 1999). For instance, consider the following operation that inserts an element at an unspecified position into a list:

```
insert :: a → [a] → [a]                                           Curry
insert x []     = [x]
insert x (y:ys) = (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]`. One can use this operation to easily define permutations:

```
perm :: [a] → [a]                                                 Curry
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 (Wadler 1985), non-deterministic operations lead to simpler program structures. Furthermore, they have 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 considerably smaller search spaces (see (Antoy and Hanus 2010; González-Moreno *et al.* 1999; Hanus 2024) for more detailed discussions).

As mentioned in Section 1, the occurrence of non-deterministic operations as arguments might yield incorrect results when such arguments are rewritten to multiple occurrences. For instance, if we evaluate the expression `xorSelf aBool` as in standard term rewriting (Baader and Nipkow 1998), that is by applying program rules from left to right, there is the derivation (among others)

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

https://doi.org/10.1017/S1471068426100453 Published online by Cambridge University Press

The result `True` is incorrect, as discussed in Section 1. It is interesting to note that this value cannot be obtained with a strict evaluation strategy where arguments are evaluated prior to the function calls. To avoid dependencies on the rewriting strategy and exclude such incorrect results, González-Moreno *et al.* (1999) 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 (Hussmann 1992), 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.

Although sharing is also used in implementations of non-strict functional languages, like Haskell, in order to support optimal evaluation (Huet and Lévy 1991), we cannot directly map functional logic programs into Haskell programs due to the non-deterministic features. Thus, a correct mapping requires modeling non-determinism *and* sharing of non-deterministic expressions. For this purpose, we will use a monadic representation of programs.

In addition to these base features, functional logic languages have more features which are useful for application programming. Apart from standard features like modules or monadic I/O (Wadler 1997), *encapsulation* (Braßel *et al.* 2004) is useful to collect all results of a non-deterministic subcomputation in some data structure, and *functional patterns* (Antoy and Hanus 2005) are useful to non-deterministically select sub-expressions at arbitrary positions. In the following, we first discuss a scheme to implement demand-driven non-determinism in Haskell. Later, we extend our scheme to more advanced features in order to obtain a full-fledged implementation of Curry in Haskell.

3 Monadic transformation

In this section we present a basic implementation of a kernel of Curry in Haskell. When mapping a functional logic program into Haskell, one has to model non-deterministic computations in a functional manner. A well-known method to represent non-deterministic results in a functional language is the "list of successes" technique (Wadler 1985): a non-deterministic operation is mapped into a function which returns a list of values. Instead of lists, one can also use other container structures, for example trees. In order to abstract from the data structure to collect values, we parameterize the target program by a monad. A monad `m` is a type constructor with two operations

```
return :: a → m a                                    Haskell
(>>=) :: m a → (a → m b) → m b
```

To model failures and non-deterministic choices, the more specific monadic structure `MonadPlus` is appropriate since it offers two additional operations

```
mzero :: m a                                         Haskell
mplus :: m a → m a → m a
```

https://doi.org/10.1017/S1471068426100453 Published online by Cambridge University Press

`mzero` represents a failing computation and `mplus` a choice between two computations. For instance, the non-deterministic operation `aBool` defined in Section 1 can be mapped into[4]

```
aBoolC :: MonadPlus m ⇒ m BoolC                                    Haskell
aBoolC = return FalseC `mplus` return TrueC
```

A simple instance of `MonadPlus` is the list monad (Wadler 1985) where `return` creates a singleton list, `mzero` an empty list, and `mplus` concatenates the argument lists. Then the monadic representation of `aBool` returns the list of its two values:

```
> aBoolC :: [BoolC]                                                Haskell
[FalseC,TrueC]
```

The monadic bind operator `(>>=)` is used to pass non-deterministic values of arguments. For instance, consider the Curry expression `not aBool`. To evaluate it, `not` must be applied to all values of `aBool`. Thus, the monadic version of `not` takes the monadic representation of its argument and returns a monadic value:

```
notC :: MonadPlus m ⇒ m BoolC → m BoolC                            Haskell
notC x = x >>= λx' → case x' of
                       FalseC → return TrueC
                       TrueC  → return FalseC
```

Since the bind operator of the list monad applies the second argument to all elements of its first argument and concatenates all result values, we can nest these operations:

```
> notC (notC aBoolC) :: [BoolC]                                    Haskell
[FalseC,TrueC]
```

Due to the monadic abstraction, one can also use other monad instances, for example search trees, or add more effects to the monadic computation. As shown later, this is the key to our implementation of advanced functional logic programming features.

Because of these considerations, we transform normal Curry code to functional code parameterized by a monad where all transformed operations get and return monadic values, which are non-deterministic computations in our case. Such a transformation for call-by-name and call-by-value languages has been presented by Wadler (1990) and is also called *monadic lifting*.

As sketched in Section 2, Curry uses a lazy call-by-need evaluation strategy. However, Haskell's sharing is not sufficient to obtain call-by-need for the monadic representation. Since the bind operator triggers the evaluation of a monadic value, multiple occurrences of the same expression might be independently evaluated. For instance, the sub-expression `aBool` of the expression `xorSelf aBool` (see Section 1) has two occurrences in the further evaluation of this expression which are independently evaluated. In order to conform to the call-time choice semantics, the non-deterministic values need to be shared. While we could evaluate `aBool` before passing it to `xor` in `xorSelf`, this would only be correct because we know `xor` to be strict. In general, such a call-by-value implementation is incorrect for a lazy language like Curry. Thus, we explicitly model sharing using an

[4] In order to distinguish Curry entities from their translations into Haskell, we decorate the latter with the suffix "`C`".

https://doi.org/10.1017/S1471068426100453 Published online by Cambridge University Press

$$[\![\forall\alpha_1 \ldots \alpha_n.\ \phi \Rightarrow \tau]\!]^t := \forall\alpha_1 \ldots \alpha_n.\ [\![\phi]\!]^t \Rightarrow \texttt{Curry}\ ([\![\tau]\!]^i) \qquad \text{(Polymorphic type)}$$

$$[\![\langle \kappa_1\ \tau_1, \ \ldots, \ \kappa_n\ \tau_n\rangle]\!]^t := \langle \mathit{rename}(\kappa_1)[\![\tau_1]\!]^i, \ \ldots, \ \mathit{rename}(\kappa_n)[\![\tau_n]\!]^i\rangle \qquad \text{(Context)}$$

$$[\![\tau_1 \to \tau_2]\!]^i := [\![\tau_1]\!]^i \to_C [\![\tau_2]\!]^i \qquad \text{(Function type)}$$

$$[\![\tau_1\ \tau_2]\!]^i := [\![\tau_1]\!]^i\ [\![\tau_2]\!]^i \qquad \text{(Type application)}$$

$$[\![\chi]\!]^i := \mathit{rename}(\chi) \qquad \text{(Type constructor)}$$

$$[\![\alpha]\!]^i := \alpha \qquad \text{(Type variable)}$$

Fig. 1. Type transformation $[\![\circ]\!]^t$.

approach adapted from both Fischer *et al.* (2011) and Petricek (2012). For this, we need to introduce an operator

```
share :: Monad m ⇒ m a → m (m a)
```
Haskell

By passing a "to-be-shared" monadic expression to `share` and extracting the result using `(>>=)`, we obtain a new monadic expression that respects our call-by-need evaluation strategy. Consider the following translation of `xorSelf` which uses its argument twice.

```
xorSelfC x = share x >>= λx' → xorC x' x'
```
Haskell

On the first evaluation of the new variable `x'` inside `xorC`, the evaluation of the original argument `x` to `share` is triggered and the computed result is memoized. On any subsequent evaluation of `x'`, the memoized result of `x` is used without evaluating `x` again.

Now we will present the transformation of Curry code to an explicit monadic variant. For the rest of this paper, we will use `Curry` to denote our monadic effect type and the following type for transformed functions to increase readability.

```
newtype a →C b = Func (Curry a → Curry b)
```
Haskell

3.1 Types

On the type level, the monadic transformation replaces type constructors with their effectful counterparts and wraps the result and argument of each function type in `Curry`. However, any type quantifiers and type constraints (which might be absent) still remain at the beginning of the type signature and we also wrap the outer type of a function. Figure 1 presents the transformation of types, including the renaming of type constructors by the operation "*rename*" (e.g., add the suffix "`C`"). For instance, the type signature

```
not :: Bool → Bool
```

is transformed into

```
notC :: Curry (BoolC →C BoolC)
```
Haskell

The transformed type signature differs from the one we gave previously for `notC`. The key difference is that we wrap the monadic type constructor `Curry` around the whole

https://doi.org/10.1017/S1471068426100453 Published online by Cambridge University Press

$$[\![\texttt{data}\ D\ \alpha_1 \ldots \alpha_n\ \texttt{=}\ C_1\ \texttt{|}\ \ldots\ \texttt{|}\ C_n\ ]\!]^d := \texttt{data}\ \mathit{rename}(D)\ \alpha_1 \ldots \alpha_n\ \texttt{=}\ [\![C_1]\!]^c\ \texttt{|}\ \ldots\ \texttt{|}\ [\![C_n]\!]^c \quad \text{(Data type)}$$

$$[\![C\ \tau_1 \ldots \tau_n]\!]^c := \mathit{rename}(C)\ [\![\tau_1]\!]^t \ldots [\![\tau_n]\!]^t \quad \text{(Constructor)}$$

Fig. 2. Data type transformation $[\![\circ]\!]^d$.

type. The latter is required because a function could introduce non-determinism before being applied to an argument. To see this, consider the following artificial function that is non-deterministically defined as either identity or negation on Boolean values.

```
idOrNot :: Bool → Bool                                    Curry
idOrNot = id ? not
```

The transformed type signature has the form

```
idOrNotC :: Curry (BoolC →C BoolC)                        Haskell
```

For this function, having the monadic type constructor `Curry` at the outer level is necessary. Since we want to decide how to transform the type of a function based solely on its type and not on its implementation, we treat all functions as potentially introducing non-determinism. Note that inlining and rewrite optimizations can get rid of some of the overhead introduced here. Such optimizations are possible with the Glasgow Haskell Compiler (GHC) that we target.

3.2 Data type declarations

Due to the non-strictness of Curry, components of data types may also contain non-determinism. As discussed in (Fischer *et al.* 2011) in detail, this can be achieved by transforming also the arguments of data constructors into monadic values. Thus, we need to modify data type definitions (except for primitive data types since they have no components). Because we rename the data types during the transformation, we have to update any type constructor in a type to use the name of its effectful counterpart. We take a look at the transformation of data types next. To support data constructors having unevaluated or non-deterministic components, we transform every constructor. As the (partial or full) application of a constructor can never be non-deterministic by itself, we neither have to transform the result type of the constructor nor wrap the function arrow. This allows us to transform only the parameters of constructors, because they are the only potential sources of non-deterministic effects in a data type. Figure 2 shows the rules for the transformation of data types. The following code shows an example for this transformation. To improve readability, we use regular algebraic data type syntax for the list type and its constructors instead of the special list syntax of Curry and Haskell. The Curry data type

```
data List  a = Nil                                        Curry
           | Cons a (List a)
```

is transformed into

```
data ListC a = NilC                                       Haskell
           | ConsC (Curry a) (Curry (ListC a))
```

https://doi.org/10.1017/S1471068426100453 Published online by Cambridge University Press

Because the transformation of a constructor differs from the transformation of a function, we later have to treat constructors and functions in expressions differently. We rename constructors and type constructors in our implementation so that we can more easily identify a transformed (type) constructor.

3.3 Functions

The type of a transformed function serves as a guide for the transformation of functions and expressions. We can derive three rules for our transformation of expressions from our transformation of types:

1. Each function arrow is wrapped in a `Curry` type. Therefore, function definitions are replaced by constants that use a sequence of lambda expressions to introduce the arguments of the original function. All lambda expressions are wrapped in a `return` because function arrows are wrapped in a `Curry` type.
2. We have to extract a value from the monad using `(>>=)` before we can pattern match on it. As a consequence, we cannot pattern match directly on the argument of a lambda or use nested pattern matching, as arguments of a lambda and nested values are effectful and have to be extracted using `(>>=)` again.
3. Before applying a function to an argument, we first have to extract the function from the monad using `(>>=)`. As each function arrow is wrapped separately, we need this extraction for every parameter applied to the original function.

Implementing this kind of transformation in one pass over a concrete Curry program is challenging. Thus, we assume that a source program is simplified first. For the remainder of this subsection, we assume that our program is already desugared into a flat form:[5] a program is a set of top-level functions in the form of lambda abstractions (nested definitions are removed by lambda lifting (Johnsson 1985)), pattern matching is represented with non-nested patterns in case expressions, all cases branches are complete (i.e., branches on missing constructors are completed with `failed`, the predefined always failing operation), and non-determinism (e.g., overlapping rules) is expressed with the choice operator `(?)`. As shown by Antoy (2001), any functional logic program or constructor-based conditional term rewriting system can be transformed into this flat form. For instance, the operation `insert` defined in Section 2 is desugared into the following flat form.

```
insert = λx → λxs → case xs of                                    Curry
  []     → [x]
  y:ys   → (x : y : ys) ? (y : insert x ys)
```

Figure 3 presents the rules of our monadic transformation on programs in flat form. We explain the rules for abstractions and applications in more detail. Most of the other rules are straightforward. Since our transformation only works for correctly typed programs, we assume that the input code to our transformation has been checked for type errors.

[5] The flat form of programs is also used for the semantics (Albert *et al.* 2005), analysis (Hanus and Skrlac 2014), and implementation (Antoy *et al.* 2020) of functional logic programs.

https://doi.org/10.1017/S1471068426100453 Published online by Cambridge University Press

$$
\begin{aligned}
[\![x]\!]^e &:= x && \text{(Variable)}\\
[\![f]\!]^e &:= \mathit{rename}(f) && \text{(Defined Function)}\\
[\![e_1\ e_2]\!]^e &:= \mathit{alias}([\![e_2]\!]^e, y, \texttt{apply}\,[\![e_1]\!]^e\ y) && \text{(Application)}\\
[\![\lambda x \to e]\!]^e &:= \texttt{return (Func } (\lambda x \to [\![e]\!]^e)) && \text{(Abstraction)}\\
[\![C]\!]^e &:= \texttt{return (Func } (\lambda y_1 \to \ldots \texttt{ return (Func } (\lambda y_n \to \\
&\qquad \texttt{return } (\mathit{rename}(C)\ y_1 \ldots y_n))) \ldots)) && \text{(Constructor)}\\
[\![\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2]\!]^e &:= \texttt{let } y \texttt{ = } [\![e_1]\!]^e \texttt{ in } \mathit{alias}(y, x, [\![e_2]\!]^e) && \text{(Let Expression)}\\
[\![\texttt{(?)}]\!]^e &:= \texttt{return (Func } (\lambda x \to \texttt{ return (Func } (\lambda y \to x \texttt{ `mplus` } y)))) \\
& && \text{(Choice operator)}\\
[\![\texttt{failed}]\!]^e &:= \texttt{mzero} && \text{(Failed computation)}\\
[\![\texttt{case } e \texttt{ of } \{br_1; \ldots; br_n\}]\!]^e &:= [\![e]\!]^e \texttt{ >>= } \lambda y \to \texttt{case } y \texttt{ of } \{[\![br_1]\!]^b; \ldots; [\![br_n]\!]^b\} \\
& && \text{(Case Expression)}\\
[\![C\ x_1 \ldots x_n \to e]\!]^b &:= \mathit{rename}(C)\ x_1 \ldots x_n \to [\![e]\!]^e && \text{(Case Branch)}\\
\\
\mathit{alias}(e_1, x, e_2) &:= \texttt{share } e_1 \texttt{ >>= } \lambda x \to e_2 && \text{(Aliasing)}
\end{aligned}
$$

Fig. 3. Expression transformation $[\![\circ]\!]^e$ (where f, y, $y_1, \ldots, y_n$ are fresh variables).

3.4 Applications

For better readability, the transformation of a function application uses the auxiliary function

```
apply mf mx = mf >>= λ(Func f) → f mx
```
Haskell

which extracts the "real" function from the monad and applies it to its monadic argument. Thus, a transformed function application applies the transformed function to the transformed and shared argument. An application of a function to more than one argument is represented as multiple nested applications. Similar to (Petricek 2012), we use `share` in the application of a function to an argument, instead of whenever a variable is brought into scope like in (Fischer *et al.* 2011). This also allows us to get rid of the "deep sharing" that Fischer *et al.* (2011) required. In contrast to (Petricek 2012), we also consider data types, as well as let and case expressions. Since constructor applications are shared via the application rule, the constructor arguments in a pattern of a case expression are already shared. Let expressions, however, introduce new variables that should be shared across different application sites in the body of the let expression. Therefore, we need to explicitly use `share` in their translation.

3.5 Lambda abstractions

A lambda abstraction is translated by wrapping it in a `return` as well as the transformed function constructor `Func` and translating the inner expression. We do not need to apply `share` to the argument of the lambda, since any argument supplied to a function is already shared by the rule for applications.

https://doi.org/10.1017/S1471068426100453 Published online by Cambridge University Press

3.6 Transformation examples

As an example, consider the operation `not` defined in Section 1. The flat form of `not` is

```
not = λx → case x of False → True                                    Curry
                     True  → False
```

This is transformed into

```
notC = return (Func (λarg → arg >>= λx →                             Haskell
                      case x of FalseC → return TrueC
                                TrueC  → return FalseC ))
```

As a further example with more complex pattern matching and applications, we show the transformation of the `perm` operation from Section 2 where we reduced some η-redexes for readability.

```
permC :: Curry (ListC a →C ListC a)                                  Haskell
permC = return (Func (λarg → arg >>= λxs →
           case xs of
             NilC       → return NilC
             ConsC y ys → share (share ys >>= apply permC) >>=
                            apply (share y >>= apply insertC) ))
```

Even though the code is now significantly more complex, the user will not have to read or write such code since the transformation can be fully automated. Another thing to note is that the applications of `share` on `y` and `ys` are superfluous in this code snippet. The reason for this and how to avoid these superfluous applications will be discussed in Section 5.4.

4 A history of monadic pull-tabbing

This section discusses some existing monadic implementations of pull-tabbing and their performance deficiencies. Our own implementation builds on some of these ideas but fixes the problems they have.

4.1 Tree-based non-determinism with fingerprinting

As already discussed in Section 3, the monadic transformation of functional logic programs supports different implementations of non-deterministic computations by providing different instances of `MonadPlus`. The list instance computes a list of all non-deterministic values and corresponds to backtracking search used in Prolog.

In order to support other search strategies, one can collect non-deterministic values in tree structures rather than lists. For this, we use a standard binary tree representation as seen below.

```
data Tree a = Empty                                                  Haskell
            | Leaf a
            | Node (Tree a) (Tree a)
```

https://doi.org/10.1017/S1471068426100453 Published online by Cambridge University Press

Thus, `Leaf` represents a single value, `Empty` no value, that is a failure, and `Node` a non-deterministic choice between trees of values. With this intuition in mind, it is straightforward to define the `Monad` and `MonadPlus` instances as follows.

```
instance Monad Tree where                                    Haskell
  return = Leaf

  Empty      >>= _ = Empty
  Leaf x     >>= f = f x
  Node t1 t2 >>= f = Node (t1 >>= f) (t2 >>= f)

instance MonadPlus Tree where
  mzero = Empty

  mplus = Node
```

Using the monadic transformation of functional logic programs with this tree monad, each operation computes a tree of non-deterministic values. By applying different tree traversals, one can easily implement different search strategies, like depth- or breadth-first search. In practice, this is used in the Curry implementation KiCS2 (Braßel *et al.* 2011) which has options to select various search strategies (e.g., depth-/breadth-first, iterative deepening, parallel) (Hanus *et al.* 2012).[6]

Using just a tree for the monadic effect is not sufficient to implement a call-time choice semantics. The tree-based monadic non-determinism yields unintended answers (as with standard term rewriting, see Section 2):

```
> xorSelfC aBoolC :: Tree BoolC                              Haskell
Node (Node (Leaf FalseC) (Leaf TrueC))
     (Node (Leaf TrueC) (Leaf FalseC))
```

Here we can observe the problem discussed in Section 1, namely the duplication of choices, which potentially leads to unsoundness w.r.t. call-time choice and duplicated computations. The former can be avoided by attaching identifiers to choices. When a choice is created by an occurrence of the operation "?", it is decorated with a fresh *choice identifier*. In a pull-tab step, that is when a non-deterministic demanded argument causes a choice of results, the choice identifier is passed from the argument to the result. In our example, all three `Node` constructors in the result tree would be decorated with the same choice identifier. If the tree traversal that extracts result values from a search tree always makes *consistent* choices, that is selects the same (left/right) branch for identically decorated choices, then the computed values are the intended ones (Antoy 2011). Therefore, implementations of functional logic languages which use pull-tabbing to support flexible and operationally complete search strategies (Braßel *et al.* 2011; Antoy and Jost 2016) often require choice identifiers in computations. A *computation branch*, which evaluates an expression with some decision for non-deterministic choices, contains a *fingerprint*, a partial mapping from choice identifiers to left/right decisions, and computes values for this fingerprint in a call-by-need manner. This is done in KiCS2 where the search tree traversal is parameterized by fingerprints (Braßel *et al.* 2011).

[6] The actual implementation of KiCS2 is not based on a monadic representation of Curry programs but uses a direct encoding of search trees in translated operations.

https://doi.org/10.1017/S1471068426100453 Published online by Cambridge University Press

Unfortunately, this method to implement non-determinism could cause a serious efficiency problem. Since pull-tabbing moves every choice occurring in arguments to the root of an expression, choices occurring in shared subexpressions are multiplied (before they are removed by fingerprinting). For instance, KiCS2 is the most efficient Curry implementation on purely functional programs (Braßel *et al.* 2011) but it might be much slower than other Curry implementations for particular uses of non-deterministic operations (Hanus 2012).

4.2 Explicit sharing of computations

A solution to pull-tabbing without fingerprinting is proposed by Fischer *et al.* (2011). Their implementation not only solves the problem of unsoundness but also aims to avoid the duplication of computations. They use the `share` operation we mentioned in Section 3 to save the result of a potentially non-deterministic computation on a heap local to the current computation branch. While their approach makes a key step in the right direction, it suffers from a flaw that decreases performance in real-world applications. The fact that results are only stored and looked up locally in the current computation branch implies that results cannot be shared across non-deterministic branches. Consider the following code snippet.

```
primes :: [Int]                                                    Curry
primes = <deterministic definition of an infinite list of primes>

sharingAcrossND :: Int
sharingAcrossND = let prime800 = primes !! 799
                  in prime800 ? prime800
```

The value of `prime800` will be computed for the first time when one argument of the choice operator `(?)` is evaluated. At that point, the value of `prime800` is stored on the heap only in that computation branch. Thus, we do not have the computed value available on the heap of the alternative branch. When evaluating the other argument of the choice, the value of `prime800` will be required again. Consequently, we need to evaluate `prime800` a second time even though it will yield the same result. In conclusion, deterministic values are not shared across non-deterministic branches in this setting, as already noticed by Fischer *et al.* (2011). It should be noted that the same problem occurs in a backtracking-based implementation of non-determinism: a computation performed in one non-deterministic branch is undone before an alternative non-deterministic branch is selected.

5 A Non-determinism monad with memoization

In this section we introduce the kernel of our implementation by developing an implementation of a basic memoized non-determinism monad that aims to fix the problems discussed in the previous section. It is based on a solution to pull-tabbing without fingerprinting that enables sharing across non-determinism, proposed in (Hanus and Teegen 2021), but here the results of non-deterministic computations are memoized in a different way so that it fits into the functional monadic transformation.

https://doi.org/10.1017/S1471068426100453 Published online by Cambridge University Press

An implementation based on the monadic transformation presented in Section 3 has to implement the functional interface on which this transformation is based, that is an appropriate monad instance and an implementation of the operation `share`. This is quite similar to the approach of Fischer *et al.* (2011) since they proposed the same interface. While they give a purely functional implementation of this interface for lazy non-determinism, their approach suffers from the mentioned drawbacks. In order to avoid these, our implementation uses a mutable (global) state to implement memoization. Thus, we implement the same interface (monad and `share`) but hide behind the interface an impure implementation for the sake of efficiency.

In standard implementations of non-strict functional languages, the node of a computation graph representing an operation is updated in-place with the computed result in order to share it. This is not possible in a functional logic language with an implementation based on pull-tabbing, since an operation might have more than one result. To overcome this problem, computation branches are uniquely identified by *branch identifiers*. Instead of updating a node in-place, potentially non-deterministic operation nodes contain a (partial) map tr, called *task result map*, from branch identifiers to results. When a branch i has to evaluate some node n containing an operation with map $n.tr$, it first checks whether $n.tr(i)$ is defined. If not, node n is evaluated and $n.tr(i)$ is set to the computed result before returning $n.tr(i)$. This evaluation scheme, called MPT (Hanus and Teegen 2021) and formally described in (Böhm *et al.* 2021), avoids the aforementioned problem of pull-tabbing. Its efficiency can compete with backtracking-based implementations (Hanus and Teegen 2021). Furthermore, it is a good basis for operationally complete implementations of functional logic languages. Existing implementations (Böhm *et al.* 2021; Hanus and Teegen 2021) use imperative target languages. In the following, we discuss a high-level Haskell-based implementation that exploits the monadic transformation introduced so far.

5.1 Memoization monad

To extend the tree-based monadic non-determinism implementation with memoization for call-by-need evaluation and call-time-choice, we need storages for memoized results and identifiers to uniquely label every branch of a non-deterministic computation. For the latter, we assume an abstract type of identifiers that support a `nextID` operation to generate new identifiers.

```
type ID                                                    Haskell
nextID :: ID → ID
```

Based on this abstract type, we define a state to store the current branch identifier and any "parent" identifier that occurred higher up in our tree of non-deterministic choices. The parents are necessary because a deterministic result that was memoized for a certain branch identifier is also valid in any child branch. Finally, we need an `IORef` to be able to generate `ID`s that are unique across all branches. The `IORef` itself will not change during computations but its content will. For example, the operation `freshID` below updates the content of this `IORef` to generate a new identifier.

https://doi.org/10.1017/S1471068426100453 Published online by Cambridge University Press

```
data MemoState = MemoState {                                    Haskell
    branchID  :: ID,
    parentIDs :: Set ID,
    idSupply  :: IORef ID
  }

{-# NOINLINE freshID #-}
freshID :: MonadState MemoState m ⇒ m ID
freshID = do
  MemoState _ _ idSupply ← get
  let val = unsafePerformIO $
              atomicModifyIORef idSupply (λi → (nextID i, i))
  return val
```

Note that we need a `NOINLINE` pragma on `freshID` so that optimizations will not interfere with our usage of `unsafePerformIO`. Instead of using unsafe features directly, we could also use the `UniqSupply` type that is used inside the GHC itself, but that one uses unsafe features under the hood as well.

Below we define the type of our basic Curry monad. It is just a state monad on top of our tree structure.

```
newtype Curry a = Curry {                                       Haskell
    unCurry :: StateT MemoState Tree a
  } deriving (Functor, Applicative, Monad)
```

The `Monad` instance can be derived from the underlying `StateT` implementation using generalized `newtype` deriving, but we explicitly need to give the `MonadPlus` instance so that we can manipulate our branch identifiers correctly.

```
instance MonadPlus Curry where                                  Haskell
  mzero = Curry (lift Empty)

  Curry m1 `mplus` Curry m2 = Curry $ do
    MemoState branchID parentIDs idSupply ← get
    i1 ← freshID
    i2 ← freshID
    let newPs = Set.insert branchID parentIDs
        leftState  = MemoState i1 newPs idSupply
        rightState = MemoState i2 newPs idSupply
    (put leftState ≫ m1) `mplus` (put rightState ≫ m2)
```

5.2 Memoizing non-determinism

Now we define the central function `share` that enables lazy evaluation via memoization.

```
share :: Curry a → Curry (Curry a)                              Haskell
```

Compared to the type described by Fischer *et al.* (2011), we have removed a type class constraint. As mentioned before, Fischer *et al.* (2011) need this additional type class to ensure that expressions nested deep inside a data structure are shared and memoized as well (referred to as "deep sharing"). However, since we use `share` at every (constructor)

https://doi.org/10.1017/S1471068426100453 Published online by Cambridge University Press

application site, we already get this deep sharing for free. The function `share` does the actual memoization work. When called with an argument computation, it first creates a new task result map to memoize possible results of this computation. Then it returns a new computation which checks and updates this map accordingly. We present the implementation of `share` step by step.

```
share :: Curry a → Curry (Curry a)                                    Haskell
share (Curry ma) = Curry $ do
  let trRef = unsafePerformIO (newIORef emptyHeap)
  MemoState b1 _ _ ← get
  return $ ... -- see below
```

To store memoized results, we use the following data type `Heap`, a mapping from `ID`s to values, with both an `insertHeap` and `lookupHeap` operation. The implementation of the heap is not relevant, as it can be any kind of key-value store.

```
type Heap a                                                           Haskell
emptyHeap  :: Heap a
insertHeap :: ID → a → Heap a → Heap a
lookupHeap :: ID → Heap a → Maybe a
```

We initialize the heap by using `unsafePerformIO` together with `newIORef emptyHeap` in the function `share` to create a kind of memory cell where we can manipulate our results.[7] In the next step, the current branch ID (`b1`) is retrieved from the state so that we can use the correct ID to insert into the task result map. This is everything that we need to do to prepare a shared computation. The actual execution of our computation `ma` happens in the returned computation which starts with the `return` in the last line. We continue at that `return`.

```
return $ do                                                           Haskell
  MemoState b2 p2 idSupply2 ← get
  case lookupTaskResult trRef b2 p2 of
    Nothing  → ... -- see below
    Just res → ... -- see below
```

Here, we retrieve the current (possibly different) state and check if we already have a result saved in the task result map that is valid in the current or a parent branch. Checking the map for a memoized result is done using `lookupTaskResult` which is defined as follows.

```
lookupTaskResult :: IORef (Heap a) → ID → Set ID → Maybe a             Haskell
lookupTaskResult trRef i p =
    msum (map (λj → lookupHeap j h) (i : Set.toList p))
  where h = unsafePerformIO (readIORef trRef)
```

The current map is read using `unsafePerformIO` and, for the current and each parent branch ID, we check if there is a valid result memoized in this map. Combining all these

[7] We tried getting rid of `IO` in our implementation but we conjecture that this is not possible for our goal. Note that the usage of `unsafePerformIO` is captured in a safe interface here and is, thus, just ugly but still safe.

https://doi.org/10.1017/S1471068426100453 Published online by Cambridge University Press

`Maybe a` results using the monoidal `msum`, we obtain the first result if it exists, or `Nothing` otherwise. Note that there can only ever be at most one valid result in the map.

Now we continue with the `Nothing` case of `share` where no valid memoized result exists.

```
Nothing → do                                                    Haskell
  a ← ma
  MemoState b3 _ _ ← get
  let wasND = b2 /= b3
  let whereTo = if wasND then b3 else b1
  let addTR h = insertHeap whereTo a h
  unsafePerformIO (modifyIORef trRef addTR)
    `seq` return a
```

Here, we have to execute the computation `ma` to obtain a result that we can memoize. Immediately afterwards, we get the current branch ID (`b3`) so that we can check if the computation `ma` contained non-determinism. If the branch ID changed during its execution (i.e., `b2 /= b3`), the computation was indeed non-deterministic and, thus, the new result is valid only for the local branch identifier `b3`. In the case that the branch ID did not change, `ma` was deterministic and, thus, the result is globally valid for the branch identifier `b1` that was active on the outer monadic layer of `share`. With this information at hand, we construct a new task result map and update the `IORef` using `unsafePerformIO`. We have to force the computation of the result of `unsafePerformIO` using `seq` to ensure that it actually happens *before* the result of the computation `ma` is returned.

If we actually have a result memoized, the `Just` branch after our lookup simply returns the result.

```
Just res → return res                                           Haskell
```

5.3 Nested sharing of non-deterministic values

The implementation given above is correct for programs where non-deterministic values are not shared twice or more. However, there are some programs where such a multiple sharing happens. To illustrate the problem, consider the following Curry program.

```
notIf :: Bool → Bool                                            Curry
notIf x = let nx = not x
          in if x then nx else nx
```

Semantically, we expect `notIf (False ? True)` to be equivalent to `True ? False`. However, with the memoization implementation given above, we get the result `True ? True`. To see why this happens, let us look at the monadic transformation of `notIf`.

```
notIfC :: Curry (BoolC →C BoolC)                                Haskell
notIfC = return (Func (λx →
           let nx' = share x >>= λx' → apply notC x'
           in share nx' >>= λnx → x >>= λx' → case x' of
                TrueC  → nx
                FalseC → nx ))
```

https://doi.org/10.1017/S1471068426100453 Published online by Cambridge University Press

Remember that `x` has already been shared during the application of `notIf` to `True ? False`. Consequently, `x` is evaluated before `nx` since `x` is demanded by `notC`. When the computation of `nx` is forced in either of the branches, the value for `x` is memoized and can just be returned for any subsequent computation without triggering any new non-determinism. Thus, the current implementation of memoization deduces that `nx` is deterministic and the value of `nx` is memoized across branches as `TrueC`. As a result, our implementation fails in situations where a shared value (`nx`) depends on another shared value (`x`) that is forced earlier than the first one.

To fix this, we also memoize whether a value is deterministic and inserted globally, or non-deterministic and inserted locally. On reading a memoized value that was non-deterministic, we simply advance the branch identifier. Thus, a repeated sharing of the same non-deterministic value or a value that directly depends on it will be assumed to be non-deterministic. In other words, we now prevent global memoization of a computation if it depends on a non-deterministic computation and not just when the computation itself introduces non-determinism. Therefore, we need to adapt the implementation. The relevant changes are in both `case`-branches.

```haskell
case lookupTaskResult trRef b2 p2 of
  Nothing  → do
    a ← ma
    MemoState b3 _ _ ← get
    let wasND = b2 /= b3
    let whereTo = if wasND then b3 else b1
    let addTR h = insertHeap whereTo (a, wasND) h
    unsafePerformIO (modifyIORef trRef addTR)
      `seq` return a
  Just (res, wasND) →
    | wasND → do
        i ← freshID

        let newParents = Set.insert b2 p2
        put (MemoState i newParents idSupply2)
        return res
    | otherwise →
        return res
```

5.4 Improving performance of sharing

There is one major performance bottleneck with this implementation. The problem is that `share` enters its argument into the memoization heap, even when a previous `share` has done so already. Repeated sharing of a computation is unnecessary and expensive in both time and memory consumption, as each `share` introduces another indirection with an `IORef`. Thus, an obvious performance optimization is to prevent repeated sharing of values as much as possible. To achieve this, we use the following idea.

We prevent unnecessary insertions of `share` during the monadic transformation whenever possible. Any locally introduced variable (bound via let, case or lambda) has already been shared by either the application or the let rule. Thus, sharing these variables again

https://doi.org/10.1017/S1471068426100453 Published online by Cambridge University Press

$$alias(e_1,x,e_2) := \begin{cases} \texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 & \text{if } e_1 \text{ is a single, locally bound variable} \\ \texttt{share } e_1 \texttt{ >>= } \lambda x \rightarrow e_2 & \text{if } e_1 \text{ is a complex expression or a globally bound variable} \end{cases}$$

(Aliasing)

Fig. 4. Improved *alias* rule.

when they are applied to a function is unnecessary. Consider the following example of a function that reverses the order of elements in a list.

```
reverse :: [a] → [a]                                    Curry
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

Here, both `x` and `xs` have already been shared when they were originally applied to the list constructor `(:)`. Thus, we can avoid sharing them explicitly. To introduce this improvement, we replace in the rules given in Figure 3 the definition of *alias* by the new definition shown in Figure 4.

6 Extending the monad

An important advantage of our monadic implementation is that one can implement new features by modifying the monad without changing the translation of source programs. In this section, we will show extensions that are relevant in contemporary functional logic languages. The first extension concerns the addition of free (logic) variables, which represent unknown values or possibly infinite sets of values. Free variables are required to support unification, that is binding free variables to partially known values instead of enumerating all their values. Thus, we will show how unification can be implemented in our monadic approach. The second extension is encapsulated search, that is a method to collect the results of non-deterministic computations in some data structure. Though this is known in logic programming for a long time (`findall`), the interaction with demand-driven computations causes new challenges which we will discuss.

6.1 Free variables

Free variables are an important feature of logic programming. They denote unknown values which are refined during a computation. It is well known that free variables can be replaced by value generators, that is non-deterministic operations that yield the (possibly infinite) set of values of a given type (Antoy and Hanus 2006). For instance, a value generator `aBool` for Boolean values was defined in Section 1. Similarly, a value generator for lists of Booleans can be defined by

```
aBoolList ::  [Bool]                                    Curry
aBoolList = [] ? (aBool : aBoolList)
```

Although the use of value generators for unknown values is sufficient from a declarative point of view, a crucial feature of logic programming is *unification* to compute with partial information, that is, without completely instantiating free variables. For instance, if `x` and `y` are free variables, the unification `x =:= y` is solved by binding `x` to `y` (or vice

https://doi.org/10.1017/S1471068426100453 Published online by Cambridge University Press

versa) without instantiating them to a value. In contrast, if `x` and `y` are generators for an infinite set of values, their evaluation might lead to an infinite search space. In order to support an efficient implementation of unification, we need to explicitly represent free variables during run time so that we can implement bindings without generating values.

For this purpose, we use the approach taken by Teegen *et al.* (2021) and adapt it to our memoization monad. Central to their approach is the idea that, if a computation demands a free variable by using `(>>=)`, the variable gets narrowed, that is it is partially instantiated. This fits to our monadic compilation scheme: whenever a value of some expression is demanded, the bind operator `(>>=)` is used, that is to extract the scrutinee of a `case`-match or to extract a function to be applied.[8] In both cases, we need an explicit value instead of a free variable in order to continue the evaluation.

Teegen *et al.* (2021) define a type class `Narrowable`, which enumerates all constructors of the given type and fills their arguments with new free variables.[9]

```
class Narrowable a where                                    Haskell
  narrow :: [Curry a]
```

Implementations for this class have to be generated for every transformed data type. For example, the instance for the list type looks as follows.

```
instance Narrowable a ⇒ Narrowable (ListC a)                Haskell
 where
  narrow = [return NilC, ConsC <*> share free <*> share free]
```

Free variables are created using the function `free` that will be introduced later. We also need to extend our state with a heap to store the bindings of free variables that have already been narrowed. Since not all free variables have the same type, our heap has to be untyped using existential data types (Perry 2005) and `unsafeCoerce` as shown below. Using unsafe features here will not cause issues at run time, as in a type-correct program every logic variable has a unique identifier and a single type.

```
data Untyped = ∀a. Untyped a                                Haskell
typed :: Untyped → a
typed (Untyped x) = unsafeCoerce x

data FreeState = FreeState {
    branchID  :: ID,
    parentIDs :: Set ID,
    varHeap   :: Heap Untyped,
    idSupply  :: IORef ID
  }
```

Now we can define the new type for our memoization monad extended with free variables. Apart from the state, it differs from our previous monad by having the type `(FLVal a)` instead of a plain `a` in its result. This new type differentiates between values

[8] The operation `(>>=)` is also used on the result of a `share`, but this is irrelevant here as `share` never yields a free variable.

[9] Our implementation of `Narrowable` is actually a bit simpler compared to (Teegen *et al.* 2021), but the reasons why that is possible do not matter here.

https://doi.org/10.1017/S1471068426100453 Published online by Cambridge University Press

and variables, with the latter one containing a `Narrowable` constraint so that we are guaranteed to be able to narrow a variable that we encounter during evaluation:

```
data FLVal a = Val a                                                    Haskell
             | Narrowable a ⇒ Var ID
```

Thus, the state of computations with free variables is defined as[10]

```
newtype CurryFree a = CF {                                              Haskell
    unCF :: StateT FreeState Tree (FLVal a)
  }
```

Now we can show the implementation of `free` that we mentioned earlier. It uses the new type `FLVal` and provides a fresh identifier of some type constraint to `Narrowable`. Here we use the `ID` type to uniquely identify free variables in addition to their use as branch identifiers.

```
free :: Narrowable a ⇒ CurryFree a                                      Haskell
free = CF $ do
  i ← freshID
  return (Var i)
```

Before we define the `Monad` instance for the `CurryFree` type, we introduce another helper function for the instantiation of free variables. This instantiation starts by generating a transformed value for each constructor of the corresponding type using `Narrowable`. For each of those values, we spawn a new computation that inserts a binding for the instantiated variable into the heap and updates the branch identifier accordingly. Thus, the `varHeap` in every branch contains a mapping from variable identifiers to monadic computations that have already been shared. We then combine the computations for each constructor by non-deterministic choices using `msum`. It is imperative that we first bind the corresponding variable and then write it on the heap so that repeated lookups of the same variable also share the same identifiers generated by calls to `free` in the `Narrowable` instance. Otherwise, two computations in the same branch could compute different identifiers for the free variables in the arguments of a narrowed constructor, leading to unsound results. We do, however, write only monadic values on the heap so that we can also put lazy computations on the heap. Thus, we wrap our instantiated value in a `CF (return ..)`.

```
instantiate :: Narrowable a ⇒ ID → CurryFree a                          Haskell
instantiate vid = CF $ do
  st ← get
  msum (map (update st) narrow)
where
 update (FreeState i ps h suppl) x = unC $ do
   i' ← freshID
   sharedX ← x
   let h' = insertHeap vid (Untyped (CF (return sharedX))) h
   put (FreeState i' (Set.insert i ps) h' suppl)
   return sharedX
```

[10] Note that we cannot derive required instances (e.g., `Monad`) automatically anymore.

https://doi.org/10.1017/S1471068426100453 Published online by Cambridge University Press

Now we can implement the `Monad` instance of `CurryFree`. The only interesting bit is that we make a case distinction on the result of the first argument so that we can check whether it is a variable or a value. Recall that we have to instantiate free variables when they are pattern matched in the source program, which corresponds to a `(>>=)` in the monadic translation. In case the result is a variable that is yet unbound, we instantiate that variable before we continue with the computation. Otherwise, we can apply the continuation from the second argument.

```
instance Monad CurryFree where                                    Haskell
  CF ma >>= f = CF $ do
    fla ← ma
    FreeState _ _ h _ ← get
    unCF $ case fla of
      Var i → case lookupHeap i h of
                Nothing → instantiate i >>= f
                Just x  → typed x >>= f
      Val x → f x
```

6.2 Monadic unification

With the extension of our monad to represent free variables, we are able to implement a standard unification procedure by inserting variable bindings directly onto the heap rather than instantiating these variables.

In the following, we show an implementation of a Prolog-like unification algorithm based on the monad described in the previous section. The important issue is the distinction between free variables and values (see type `FLVal` defined above). If free variables are unified, a binding between these variables is stored onto the heap instead of instantiating them.

We start by defining a type class `Unifiable` which contains the operation `unify`.

```
class Unifiable a where                                           Haskell
  unify :: a → a → CurryFree BoolC
```

Based on `unify`, we define a generic operation `unifyC` which checks whether the arguments are free variables or values. Before the check, we have to follow any chain of variables with `deref` since it can happen that a variable is bound to another one. If both arguments are unbound free variables, we bind one variable to the other without instantiating any of them. In the case that only one of the arguments is a variable, we instantiate it and unify recursively. Whenever both arguments are already instantiated, we simply unify their values.

```
unifyC :: Unifiable a                                             Haskell
       ⇒ CurryFree a → CurryFree a → CurryFree BoolC
unifyC ma mb = CF $ do
  fla ← deref ma
  flb ← deref mb
  unCF $ case (fla, flb) of
  (Var i, Var j) → do
```

https://doi.org/10.1017/S1471068426100453 Published online by Cambridge University Press

```
    FreeState b ps h suppl ← get
    let h' = insertHeap i (Untyped (CF (return (Var j)))) h
    put (FreeState b ps h' suppl)
    return TrueC
  (Var i, Val y) → instantiate i >>= unify y
  (Val x, Var j) → instantiate j >>= unify x
  (Val x, Val y) → unify x y
where
  deref (CF m) = do
    fl ← m
    case fl of
      Var i → get >>= λ(FreeState _ _ h _) →
              case lookupHeap i h of
                Just x  → deref (typed x)
                Nothing → return (Var i)
      Val x → return (Val x)
```

The instances of `Unifiable` can be schematically generated together with the transformation of data types. Note that `unify` either returns `TrueC`, if both arguments are unifiable, or fails. The instances for base types are easy, as shown for the type of Booleans.

```
instance Unifiable BoolC where                                    Haskell
  unify FalseC FalseC = return TrueC
  unify TrueC  TrueC  = return TrueC
  unify _      _      = mzero
```

Structured types are unified by pairwise unifying the arguments of identical data constructors, as shown for the list data type.

```
instance Unifiable a ⇒ Unifiable (ListC a)                        Haskell
 where
  unify NilC         NilC         = return TrueC
  unify (ConsC x xs) (ConsC y ys) = (&&) <$>
                                    unifyC x y <*> unifyC xs ys
  unify _            _            = mzero
```

Finally, Curry's unification operator, as introduced in Section 2, can be implemented as follows.

```
(=:=) :: Unifiable a ⇒ CurryFree (a →C a →C BoolC)                Haskell
(=:=) = return $ Func $ λx →
          return $ Func $ λy → unifyC x y
```

The explicit instances of `Unifiable` for each data type might look cumbersome but, actually, it is a feature of our approach. All these instances can be schematically generated by a compiler. More important, it is not reasonable to have instances for all types. Hanus and Teegen (2020) proposed the introduction of a type class `Data` which supports value generators, strict equality, and unification for data types. Since functional types have no `Data` instances, the operators supported by the `Data` class are overloaded so that a different implementation is required for each data type, as in our approach.

https://doi.org/10.1017/S1471068426100453 Published online by Cambridge University Press

6.3 Encapsulated search

Non-deterministic programming with built-in search supports a compact and elegant programming style, as known from logic programming and also used in functional logic programming (Antoy and Hanus 2002). In application programs, it is necessary to encapsulate non-deterministic computations so that the outcomes are collected in some data structure, for example to print the results in a deterministic manner or to select a result according to some criterion. Prolog offers various operations for this purpose, like `findall` and similar predicates. However, a direct transfer of such predicates to a language with a demand-driven evaluation strategy is problematic. For instance, `findall` always computes all solutions so that it cannot be applied to infinite search spaces. If an encapsulation operator would return a possibly infinite list or tree, lazy evaluation allows to process part of this structure in a finite computation. Therefore, proposals for encapsulated search in functional logic languages try to support demand-driven evaluation of non-deterministic computations, for example (Antoy and Braßel 2007; Antoy and Hanus 2009; Christiansen *et al.* 2013). For instance, if `allValues e` denotes the list of all values of the expression e, one can test whether e has no value by `null (allValues e)`. Thanks to lazy evaluation, one gets a Boolean result even if e has infinitely many values.

Unfortunately, the combination of encapsulated search and demand-driven evaluation causes new challenges. A potential problem occurs if the encapsulated expression e shares subexpressions introduced outside the encapsulation operator. For instance, consider the expression

```
let b = False ? True in allValues (not b)
```
Curry

The obvious question is whether the non-determinism of `b` should be encapsulated by `allValues` or not. There is no clear answer but there are different views. *Strong encapsulation*, inspired by Prolog's `findall`, requires encapsulating all non-determinism accessible from the encapsulated expression so that this expression yields the single list `[True,False]`.[11] Unfortunately, the results of strongly encapsulating search operators depend on the evaluation strategy. If we consider the expression

```
let b = False ? True in (allValues (not b), b)
```
Curry

the computed pairs of values depend on the order of evaluating the components of the pair. If they are evaluated from left to right, we obtain the list of values

```
[([True,False], False), ([False,True], True)]
```
Curry

With a right-to-left evaluation, we obtain `[([True], False), ([False], True)]` since, due to sharing, the already evaluated values of `b` are encapsulated. These and more pitfalls of strong encapsulation are discussed (Braßel *et al.* 2004).

One can avoid these pitfalls by a clear distinction between expressions evaluated *inside* an encapsulation operator, that is where all values of these expressions are collected, and

[11] One could also return a multi-set instead of a list in order to avoid enforcing an ordering on the solutions.

https://doi.org/10.1017/S1471068426100453 Published online by Cambridge University Press

outside, where different values lead to different results. For this purpose, Antoy and Hanus (2009) proposed *set functions* which encapsulate the non-determinism caused by a function but do not encapsulate the non-determinism of arguments. It has been shown that set functions are a strategy-independent notion of encapsulating search in demand-driven non-deterministic languages.

Instead of discussing more details about various search operators, we want to show how we can implement strong encapsulation (other search operators can be based on this by distinguishing the subexpressions to be encapsulated). In the following, we will see a further advantage of our monadic approach. Instead of adapting the compilation scheme to support encapsulated search, as done in other Curry compilers (see Section 9), we use the same monadic target programs but put all the extensions inside the implementation of the monad.

A crucial step for strong encapsulation in a lazy language is reducing an arbitrary expression to a fully-evaluated, constructor-based value. To do this for arbitrary data types, we define a type class `NormalForm` and an additional operation `normalFormCurry`.[12]

```
class NormalForm a where                                        Haskell
  normalForm :: a → Curry a

normalFormCurry :: NormalForm a ⇒ Curry a → Curry a
normalFormCurry ma = ma >>= normalForm
```

Since the arguments of an arbitrary data type might contain non-deterministic computations, the result of `normalForm` can be non-deterministic. Conceptually, the operation "pulls" the non-determinism to the outside of a data structure, which is why this is also known as pull-tabbing.

For our list data type, an instance looks as follows.

```
instance NormalForm a ⇒ NormalForm (ListC a) where              Haskell
  normalForm NilC          = return NilC
  normalForm (ConsC x xs) = do
    x' ← normalForm x
    xs' ← normalForm xs
    return (ConsC (return x') (return xs'))
```

Using this type class, `allValues` can be defined by evaluating the normal form of the encapsulated expression with the current state, collecting all results using some tree traversal (here: depth-first), and converting the Haskell-list into the internal Curry-list representation.

```
allValues :: NormalForm a ⇒ Curry a → Curry (ListC a)           Haskell
allValues ma = Curry $ get >>= λstate →
  let tree = evalStateT (unCurry (normalFormCurry ma)) state
  in return (toCurryList (dfs tree))
```

[12] Our actual compiler uses a version of the given class that is overloaded in the recursive call. This enables the definition of a normal form computation that instantiates free variables and one that does not

https://doi.org/10.1017/S1471068426100453 Published online by Cambridge University Press

The depth-first tree traversal and list conversion are simply defined as:

```
dfs :: Tree a → [a]                                                    Haskell
dfs Empty      = []
dfs (Leaf x)   = [x]
dfs (Node a b) = dfs a ++ dfs b

toCurryList :: [a] → ListC a
toCurryList []     = NilC
toCurryList (x:xs) = ConsC (return x) (return (toCurryList xs))
```

An advantage of the tree representation for non-determinism is that we can replace the depth-first tree traversal by other ones, for example breadth-first or iterative deepening. An interesting alternative is an implementation of a *fair search* (Böhm *et al.* 2021) as described in the following.

6.4 Fair search

Consider the following non-deterministic definition.

```
sometimesLoops :: Bool                                                 Curry
sometimesLoops = loop ? True ? loop
  where loop = loop
```

Conceptually, `sometimesLoops` has exactly one result. However, both a depth-first search and a breadth-first search will loop forever without ever finding the value `True` due to the non-terminating computation in the left and/or right branch. As an alternative, a *fair search* strategy should ensure that all results are computed in a finite amount of time, even in the presence of non-terminating computations in non-deterministic branches.

One can implement a fair search strategy in Haskell by traversing non-deterministic branches of the search tree in concurrent threads where leaf values are written into an answer channel. A basic version of this idea can be implemented with the concurrency features of Haskell (Peyton Jones *et al.* 1996).

```
fsBasic :: Tree a → [a]                                                Haskell
fsBasic t = unsafePerformIO $ do
  ch ← newChan
  let go t' = case t' of
                Empty    → return ()
                Leaf x   → writeChan ch x
                Node l r → do
                  _ ← forkFinally (go l) (λ_ → return ())
                  _ ← forkFinally (go r) (λ_ → return ())
                  return ()
  _ ← fork(go t)
  getChanContents ch
```

This implementation performs the following steps:

1. Create a new channel `ch` for communicating result values.
2. Define a recursive function `go` to traverse the tree where

https://doi.org/10.1017/S1471068426100453 Published online by Cambridge University Press

- leaf values are directly written in the channel `ch`, and
- concurrent threads are spawned for left and right subtrees.

3. Run `go` concurrently and collect results by using `getChanContents`.

While this fair search will spawn a huge amount of threads, these are all relatively lightweight and directly managed by GHC's run-time system.

A disadvantage of this simple approach is that the channel stays open, even when all threads have already completed their work. To determine when all threads have finished their work, we use Haskell's mutable variables `MVar` (Peyton Jones *et al.* 1996) as a semaphore to compensate the lack of a *join* operation on threads. If a thread tries to read an empty `MVar`, it suspends until it is eventually filled by other threads. When all threads have completed their work, the outermost thread will write a `Nothing` value into the channel to signal completion. Since `getChanContents` returns the results lazily, taking values until we find `Nothing` is also lazy enough.

```
fs :: Tree a → [a]                                                    Haskell
fs t = unsafePerformIO $ do
  ch ← newChan
  let go t' = case t' of
                Empty    → return ()
                Leaf x   → writeChan ch (Just x)
                Node l r → do
                  (mR, mL) ← (,) <$> newEmptyMVar <*> newEmptyMVar
                  _ ← forkFinally (go l) (λ_ → putMVar mL ())
                  _ ← forkFinally (go r) (λ_ → putMVar mR ())
                takeMVar mL ≫ takeMVar mR
_ ← forkFinally (go t) (λ_ → writeChan ch Nothing)
catMaybes ∘ takeWhile isJust <$> getChanContents ch
```

One further optimization implemented in our compiler is the use of *finalizers* to "hook" onto the garbage collector based on the concepts described by Peyton Jones *et al.* (2000). When the garbage collector cleans up the result list of our fair search, we know that no more elements will be taken out of the channel. Thus, we can stop all threads that are still active. For instance, if we demand only a single value from a search tree with more non-deterministic branches, the remaining threads are automatically terminated after getting a value and garbage collecting the list of possible values.

6.5 Further extensions

As mentioned in Section 2, there are still some other extensions that have been proposed for high-level declarative programming. The most relevant are *functional patterns* (Antoy and Hanus 2005) where one can use defined functions in addition to data constructors in patterns. Functional patterns support a compact programming style since a functional pattern abbreviates a (possibly infinite) set of standard patterns. This allows for deep pattern matching which is exploited in (Hanus 2011) to process XML structures.

For instance, the operation `last`, shown in Section 2 in a classical functional logic programming style, can be defined with a functional pattern by

```
last (_ ++ [x]) = x                                                   Curry
```

https://doi.org/10.1017/S1471068426100453 Published online by Cambridge University Press

Although this is basically the specification of the property to be satisfied by `last`, it is also executable if functional patterns are supported and even yield results if the list contains failing computations before the last element. Conceptually, a functional pattern denotes all regular patterns to which it can be evaluated. In our example, these are all lists with `x` as a final element. Antoy and Hanus (2005) showed that functional patterns can be implemented by a non-strict unification procedure where pattern variables are not fully evaluated, in contrast to the operator `unify` described in Section 6.2. Since the necessary changes are limited, we skip a detailed description of them.

The full implementation of our monad and compiler with all extensions described above is available in our GitHub repository https://github.com/Ziharrk/kmcc.

7 Optimizations for deterministic expressions

Not all operations defined in a Curry program make use of non-determinism and other logic features. In fact, only certain parts of a program are written in a non-deterministic style. These parts are usually encapsulated (see Section 6.3) and their results are then further processed by purely functional or IO-based code. To optimize the performance of our compilation model, we can identify these deterministic parts and essentially keep them as-is in the target Haskell code. As an example, consider the following definitions.

```
permutations123 :: [Int]                                    Curry
permutations123 = perm [1,2,3]

permutationLengths :: [Int]
permutationLengths = map length (allValues permutations123)
```

The functions `map` and `length` are both deterministic. Thus, computing the size of a list of permuted values should only require a non-deterministic computation for the permutation itself, whereas the remaining part of the computation is purely functional.

Deterministic parts of a program can be identified with a simple static analysis using a fixed-point computation. In the following, we will focus on using this information to optimize the compilation of deterministic parts of a program.

7.1 Deterministic sub-computations

In Section 3 we have shown that data types are transformed into a variant where each constructor argument is a monadic computation. However, in order to keep deterministic parts of a program as-is, we need to keep a deterministic copy of each function together with the original data types. When the result of a deterministic function is passed into a non-deterministic operation, we have to convert the plain representation into the monadic representation of the given data type. These conversions can be easily generated for each data type. In our implementation, they are part of a type class `FromHs`. Since our target language Haskell is typed, we also need a connection between these data types on the type level. For this purpose, we use an injective type family (Eisenberg *et al.* 2014) which maps each Curry type to its Haskell representation. The corresponding class and type family definition as well as an example instance for lists are shown below.

https://doi.org/10.1017/S1471068426100453 Published online by Cambridge University Press

```
type family HsEquivalent (a :: k) = (b :: k) | b → a                    Haskell

class FromHs a where
  from :: HsEquivalent a → a

fromHs :: HsEquivalent a → Curry a
fromHs = return ∘ from

type instance HsEquivalent ListC = List
instance FromHs a ⇒ FromHs (ListC a) where
  from Nil          = NilC
  from (Cons x xs) = ConsC (fromHs x) (fromHs xs)
```

With this conversion, the optimized transformation of the function `permutation123` from the example above can be implemented as follows.

```
permutations123 :: Curry (ListC Int)                                    Haskell
permutations123 = permC (fromHs [1,2,3])
```

There is just one catch with this optimization in the general setting: functions as arguments. If a deterministic function is passed into a non-deterministic one, we need to convert the function into its monadic representation. However, this is not possible in general: the converted function has to operate on monadic values but the function we are trying to convert operates on pure values. One could encapsulate the non-deterministic argument values and map the original function over all these values, but this approach is too strict in certain cases. Thus, our optimization needs to ensure that no deterministic function is passed as an argument to a non-deterministic one. This can be a bit challenging to implement since a function could be "hidden" inside a data type. In any of these cases, we simply fall back to the non-deterministic variants of the involved functions and data types.

A further complication comes from Curry's demand-driven evaluation strategy. A deterministic function might trigger non-determinism when forcing the evaluation of one of its arguments. Thus, each argument of a deterministic function needs to be deterministic so that the optimized version is applicable.

Due to these complications, a purely static approach would often use non-deterministic variants of functions where it is not necessary at run time. Without the ability to detect deterministic values at run time, once we enter the realm of non-determinism, there is no turning back.

7.2 Marking deterministic values

In order to avoid the problems mentioned above, we can mark a value as deterministic at run time. To do this, we extend each data type by a new constructor with a single argument that contains the deterministic variant of the given type. The updated transformation rule for data types is shown in Figure 5. For the `List` type, the final definition of its non-deterministic variant is as follows.

```
data ListC a = NilC                                                     Haskell
             | ConsC (Curry a) (Curry (ListC a))
             | ListC# (HsEquivalent (List a))
```

https://doi.org/10.1017/S1471068426100453 Published online by Cambridge University Press

$$[\![\texttt{data}\ D\ \alpha_1 \ldots \alpha_n\ \texttt{=}\ C_1\ \texttt{|}\ \ldots\ \texttt{|}\ C_n\ ]\!]^d := \texttt{data}\ \mathit{rename}(D)\ \alpha_1 \ldots \alpha_n\ \texttt{=}\ [\![C_1]\!]^c\ \texttt{|}\ \ldots\ \texttt{|}\ [\![C_n]\!]^c \\ \texttt{|}\ D\texttt{\#}\ (\texttt{HsEquivalent}\ (D\ \alpha_1 \ldots \alpha_n)) \quad \text{(Data type)}$$

Fig. 5. Revised data type transformation $[\![\circ]\!]^d$.

$$[\![C\ x_1 \ldots x_n \to e]\!]^b := \mathit{rename}(C)\ x_1 \ldots x_n \to [\![e]\!]^e\,; \\ D\texttt{\#}\ (C\ x_1 \ldots x_n) \to (\texttt{..}) \quad \text{(Case Branch)}$$

Fig. 6. Revised transformation rule for branches of case expressions $[\![\circ]\!]^b$.

In some cases it can be convenient to have a dual function to `fromHs` which receives a value in its monadic representation and returns a computation that yields the deterministic representation of the given value. Our compiler uses this function for some foreign function implementations but it is not further relevant to this paper.

Since we added another constructor to each data type, our previous transformation for case expressions becomes incomplete. We need to add a case for the new constructor where we exploit the fact that the argument is known to be deterministic. Since it is easier to define, we duplicate each branch of the case expression, where the new branch pattern includes the new constructor. The revised transformation rule for branches of case expressions is given in Figure 6. It omits the right-hand side of the new branch since it is hard to capture the decision of whether and how to use the deterministic variant of a function in a compact way. The decision is based on the used in-scope variables and their determinism status. An example of the new transformation is shown below for the `length` function, where `succ` is the increment function.

```
lengthC :: Curry (ListC a →C IntC)                                    Haskell
lengthC = return (Func (λxs →
  case xs of
    NilC                → return 0
    ListC# Nil          → return 0
    ConsC _ xs          → share (apply lengthC xs) >>= apply succC
    ListC# (Cons _ xs)  → return (succ (length xs)) ))
```

Note that one of the branches exclusively uses the deterministic function for the recursive call. If we were to assume that `length` is non-deterministic, the last branch would look as follows.

```
ListC# (Cons _ xs') → let xs = fromHs xs' in                          Haskell
                      succC >>= λf → f (lengthC xs)
```

One might notice that doubling the number of branches can lead to an exponential increase in the size of the generated code in case of deeply nested case expressions. Actually, this turned out to be practically relevant since there are programs where the flat form yields large case expressions (e.g., pattern matching on strings). This can be avoided by transforming programs so that there is at most one case expression directly at the root of the function. This form of programs, which are also called *uniform*, can be obtained by moving nested case expressions into new functions (Moreno-Navarro *et al.* 1990). The transformation into uniform programs is also used in other Curry compilers (Braßel *et al.*

https://doi.org/10.1017/S1471068426100453 Published online by Cambridge University Press

2011; Hanus *et al.* 2025). With uniform programs, we do not need to double the number of branches in case expressions but only add a single one for the new constructor where we call the appropriate version of the same function we are already in. This allows us to still "switch" to the deterministic world without an exponential code increase.

One nice additional benefit of the new constructor is that we can use it to defer the conversion of a value in the `FromHs` class. Instead of converting a value to its monadic representation immediately, we can just wrap it in the new constructor. This way, we can still pass the value or its sub-components to the deterministic variant of a function.

8 Evaluation

The objective of this work is to create a high-level memoization implementation for Curry based on a purely functional interface. This implementation is intended to support sharing of values to avoid repeated pull-tab steps and to allow for sharing of deterministic computations even across non-determinism.

Due to the monadic abstraction, the implementation of our compiler allowed for a clean separation between the run-time system (i.e., the monad and `share` implementation) and the monadic transformation. The full compiler implementation that we have now shows that the code size is smaller (thus, better maintainable) and more modular than KiCS2 (Braßel *et al.* 2011) which also compiles into Haskell.

In order to evaluate the overall efficiency of our high-level implementation, we compare it on various benchmark programs[13] with three other major Curry implementations. PAKCS (Hanus *et al.* 2025) compiles to Prolog (SWI-Prolog 9.0.4) so that its search strategy is based on backtracking. KiCS2 (Braßel *et al.* 2011) compiles to Haskell (GHC 9.4.5) and implements non-determinism by pure pull-tabbing without memoization. Curry2Go (Böhm *et al.* 2021) compiles to Go (1.19.3) and uses an imperative implementation of memoization. All benchmarks were executed on a Linux machine running Debian 12 with an Intel Core i7-7700K (4.2 GHz) processor with eight cores. We measured the average elapsed time (in seconds) of three runs using the `time` command and the executables generated by the respective compilers.

Figure 7 shows the timings for various programs. The first seven benchmarks test the properties that our implementation was designed for, while the last two benchmarks are purely deterministic programs. The remaining two benchmarks test expensive non-deterministic computations. Times for our approach are both "Monadic MPT" columns, where the latter one integrates the optimization for fully deterministic sub-computations from Section 7. Both are performed with a depth-first search strategy, GHC optimizations (`-O1`) and its single-threaded RTS. We will focus on the non-optimized version of "Monadic MPT" first and discuss the optimized implementation at the end of this section. The benchmarks are based on the ones used by Böhm *et al.* (2021) to evaluate MPT.

Compared to our initial prototype which was based on a direct integration into GHC using a compiler plugin (Hanus *et al.* 2022), the unoptimized implementation of our compiler lost a lot of performance. This seems to be the case for benchmarks with functions

[13] The benchmarks are available in our GitHub repository.

https://doi.org/10.1017/S1471068426100453 Published online by Cambridge University Press

Program	Monadic MPT	Monadic MPT + det. optim.	KiCS2	PAKCS	Curry2Go
`addNum5`	2.86	2.82	4.36	0.33	0.43
`addNum10`	4.29	4.25	13.70	0.41	0.68
`select50`	0.08	0.02	0.34	0.26	0.05
`select100`	0.57	0.04	5.27	0.27	0.07
`select150`	1.90	0.09	28.43	0.30	0.18
`yesSharingND`	1.33	0.02	0.42	33.65	4.62
`noSharingND`	2.58	0.02	0.78	34.07	2.97
`permSort`	2.55	2.46	2.80	10.30	5.96
`sortPrimes`	7.67	0.01	0.03	98.56	1.02
`naiveReverse`	3.52	0.14	0.20	6.46	1.15
`queens10`	221.26	0.09	0.45	185.31	27.41

Fig. 7. Timings (in seconds) of various programs evaluated with different compilers, green marks best time.

that are "more strict" in their arguments, suggesting that GHC's demand analysis was more effective for the plugin implementation. However, our optimized implementation reaches the same performance as the prototype for all but one benchmark.

The first five benchmarks check memoization, that is the avoidance of repeated pull-tabbing. The contrived programs `addNum5` and `addNum10` test this property by non-deterministically generating a number `x` between `0` and `2000` (inclusive, see `someNum` below) and adding `x` five and ten times to itself, respectively.

```
someNum :: Int → Int                                                    Curry
someNum n | n <= 0    = 0
          | otherwise = n ? someNum (n-1)

addNum5 :: Int → Int
addNum5 n = let x = someNum n in x+x+x+x+x
```

With MPT, the choice for `x` should be made only once and not on each addition of `x`. Looking at the times in Figure 7, our implementation requires for `addNum10` less than double the time of `addNum5`. This is in contrast to KiCS2 which does not use memoization to implement pull-tabbing. Compared to our initial prototype, this is the only benchmark where our optimized implementation does not perform with the best timings. We attribute this to the fact that the integration of the prototype into GHC allowed for more aggressive optimizations, especially for numeric-heavy computations. It should be noted that PAKCS and Curry2Go, which are faster on this benchmark, use a direct representation of integers which do not support searching on integer variables (similarly to Prolog), whereas our implementation allows for searching and constraint solving on integers via Z3 (de Moura and Bjørner 2008).

The next three benchmarks `select`n non-deterministically select an element in a list of length n and sums up the element and the list without the selected element. They also demonstrate the advantage of MPT compared to the "raw" implementation of pull-tabbing in KiCS2.

https://doi.org/10.1017/S1471068426100453 Published online by Cambridge University Press

The benchmarks `yesSharingND` and `noSharingND` use the `primes` example from Section 4.2 to create an expensive deterministic computation that yields the 800th prime number.

```
prime800 :: Int                                                    Curry
prime800 = primes !! 799
```

This computation is used in the following tests, where the first one shares the value of `prime800` through a `let`-binding across the non-deterministic choice and the second one does not.[14]

```
yesSharingND :: Int                                                Curry
yesSharingND = let p = prime800 in p ? p

noSharingND :: Int
noSharingND = prime800 ? prime800
```

Since `noSharingND` takes approximately twice as long as `yesSharingND`, we can conclude that our implementation shares deterministic values across non-determinism.

The final benchmarks are taken from (Hanus and Teegen 2021). `permSort` sorts a list of 13 elements by non-deterministically generating all permutations and keeping the sorted one. `sortPrimes` generates prime numbers as shown above and sorts a list of four of them using permutation sort. This benchmark mixes determinism and non-determinism. The large execution time of PAKCS is due to the fact that it does not implement sharing across non-determinism so that the prime numbers are recomputed in each permutation. `naiveReverse` is a simple deterministic example of the quadratic algorithm to reverse a list of 4096 elements, and `queens10` computes the number of safe positions to put 10 queens on a 10×10 chessboard using Peano numbers.

These benchmarks indicate that, for purely functional programs, our non-optimized implementation is faster than PAKCS and sometimes Curry2Go but slower than KiCS2. For non-deterministic programs, the results depend on the exact benchmark. Since our implementation optimizes for programs such as `addNum10` or `select150` where avoiding repeated pull-tabbing is relevant, we are predictably faster than KiCS2. PAKCS is sometimes faster since it uses backtracking in Prolog instead of pull-tabbing.

Unfortunately, the non-optimized version is slower than Curry2Go and much slower than KiCS2 in the mixed benchmark `sortPrimes`. While this may seem surprising at first, the speed of KiCS2 stems from the fact that it also optimizes deterministic operations, even when deterministic functions are applied to potentially non-deterministic arguments. This is actually very beneficial in the implementation of `sortPrimes` and for the two deterministic benchmarks as well. Without this optimization, KiCS2 is about as slow as our unoptimized approach.

Using the optimizations for deterministic computations presented in Section 7, we can achieve a similar performance to KiCS2 and sometimes even outperform it in the purely functional benchmarks. This is due to the fact that for those functions our compiler (together with GHC) essentially generates the same code as GHC would have generated if we had compiled the benchmark as a Haskell program. However, since our implementation

[14] Note that top-level declarations are always operations in Curry. Their results are never shared.

https://doi.org/10.1017/S1471068426100453 Published online by Cambridge University Press

fundamentally differs from the one in KiCS2, it is still possible to construct programs where KiCS2 outperforms our compiler.

9 Related work

There are two different groups of related works which we discuss in the following subsections.

9.1 Other compilers for Curry

While we have already mentioned other Curry compilers throughout the paper, here we summarize similarities and differences between them and our approach.

The compiler most similar to our work is KiCS2 (Braßel *et al.* 2011). It compiles to Haskell and the evaluation strategy is based on pull-tabbing as well. However, KiCS2 does not support memoization. While its model of Curry programs is not directly monadic, it bears some resemblance. Instead of having an explicit effect data type like `Tree` in our approach, the structure is basically inlined by augmenting every data type with a constructor for non-deterministic choices and for failure. Instead of using a state monad to pass around information, KiCS2 augments every function with corresponding parameters. This allows better optimization but results in more complex and less maintainable code. For instance, the implementation of set functions in KiCS2 requires a modification in the compilation scheme by adding an additional argument (the encapsulation level) to each function (Christiansen *et al.* 2013).

Prott *et al.* (2023) uses the same basic idea of a monadic transformation to augment the GHC with different semantics, including one with Curry-style non-determinism. However, that implementation lacks the performance improvements and concentrates on the integration into the Haskell compiler itself.

Other related compilers are Curry2Go (Böhm *et al.* 2021) and its Julia-based predecessor (Hanus and Teegen 2021). These compilers introduced and refined the memoization approach to pull-tabbing but both used an imperative language as their back end. Thus, the compilation scheme and their modeling of Curry are substantially different.

Sprite (Antoy and Jost 2016) compiles Curry to LLVM in order to generate efficient target code. It is also based on pull-tabbing but does not implement memoization. We could not include a comparison in our benchmarks since a working implementation is not available. The results published in (Antoy and Jost 2016) indicate that the performance of Sprite behaves similarly to that of KiCS2.

The PAKCS compiler from Curry to Prolog (Hanus *et al.* 2025) is very different from our approach. Due to the use of Prolog as a target language for compilation, PAKCS inherits the incompleteness of backtracking as a search strategy.

9.2 Monadic intermediate languages

Our transformation basically models the denotational semantics of Curry explicitly. Peyton Jones *et al.* (1998) have applied such an approach to design a common monadic intermediate language for Haskell and ML. Although we do not use our monadic intermediate language to model two languages, the idea remains the same.

https://doi.org/10.1017/S1471068426100453 Published online by Cambridge University Press

Transforming an effectful language into purely functional code using a monadic style has been used in the past to model functional languages in various proof assistant systems. For instance, Abel *et al.* (2005) generate Agda code that models Haskell's semantics via an explicit monadic effect.

More recently, a compilation scheme for the algebraic effect language *Eff* has been presented that translates Eff into monadic OCaml (Karachalias *et al.* 2021). However, Eff is an entirely different language than Curry with different challenges.

10 Conclusions and future work

In this work, we developed an implementation of Curry which supports various advanced features of functional logic languages introduced in recent years. In order to achieve a sufficient evaluation performance even when combining lazy evaluation with non-determinism, we adapted the recent method of memoized pull-tabbing from Curry compilers that use imperative target languages to a functional language. This evaluation strategy is modeled in a monadic style and can be combined with an automatic transformation of Curry programs into monadic Haskell programs. We also extended this implementation to efficiently support free variables from Curry and integrated other extensions, like unification, functional patterns, encapsulated search, or fair search. Due to the monadic structure of the target programs, these extensions can be implemented by modifying the monad only without changing the compilation scheme. The results for small but typical benchmarks indicate a good performance compared to other compilers. Thus, we obtained a high-level, maintainable, and efficient implementation of Curry which supports all language features together with an operationally complete, fair search strategy.

For future work, there is a lot to do. Our determinism analysis and optimization would of course be improved by user-given annotations in the source language, that is enforcing that certain type class functions cannot be non-deterministic. Additionally, using a strictness analysis would also allow us to omit certain `share` usages and omit laziness when it is not necessary.

Competing interests

The authors declare none.

References

Abel, A., Benke, M., Bove, A., Hughes, J. and Norell, U. 2005. Verifying Haskell programs using constructive type theory. In *Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell 2005*, ACM Press, New York, NY, USA, 62–73.

Albert, E., Hanus, M., Huch, F., Oliver, J. and Vidal, G. 2005. Operational semantics for declarative multi-paradigm languages. *Journal of Symbolic Computation* 40, 1, 795–829.

Alqaddoumi, A., Antoy, S., Fischer, S. and Reck, F. 2010. The pull-tab transformation. In *Proceedings of the Third International Workshop on Graph Computation Models 2010*, Enschede, The Netherlands, 127–132. Published Online. Available at http://gcm2010.imag.fr/pages/gcm2010-preproceedings.pdf

https://doi.org/10.1017/S1471068426100453 Published online by Cambridge University Press

Antoy, S. 1997. Optimal non-deterministic functional logic computations. In *Proceedings of the International Conference on Algebraic and Logic Programming (ALP'97) 1997*, Springer LNCS, Berlin, Heidelberg, 1298, 16–30.

Antoy, S. 2001. Constructor-based conditional narrowing. In *Proceedings of the 3rd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2001) 2001*, ACM Press, New York, NY, USA, 199–206.

Antoy, S. 2011. On the correctness of pull-tabbing. *Theory and Practice of Logic Programming* 11, 713–730.

Antoy, S. and Braßel, B. 2007. Computing with subspaces. In *Proceedings of the 9th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'07) 2007*, ACM Press, 121–130.

Antoy, S., Echahed, R. and Hanus, M. 2000. A needed narrowing strategy. *Journal of the ACM* 47, 4, 776–822.

Antoy, S. and Hanus, M. 2000. Compiling multi-paradigm declarative programs into Prolog. In *Proceedings of the International Workshop on Frontiers of Combining Systems (FroCoS'2000) 2000*, Springer LNCS, Berlin, Heidelberg, 1794, 171–185.

Antoy, S. and Hanus, M. 2002. Functional logic design patterns. In *Proceedings of the 6th International Symposium on Functional and Logic Programming (FLOPS 2002) 2002*, Springer LNCS, 2441, 67–87.

Antoy, S. and Hanus, M. 2005. Declarative programming with function patterns. In *Proceedings of the International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'05) 2005*, Springer LNCS, Berlin, Heidelberg, 3901, 6–22.

Antoy, S. and Hanus, M. 2006. Overlapping rules and logic variables in functional logic programs. In *Proceedings of the 22nd International Conference on Logic Programming (ICLP 2006) 2006*, Springer LNCS, Berlin, Heidelberg, 4079, 87–101.

Antoy, S. and Hanus, M. 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) 2009*. ACM Press, New York, NY, USA, 73–82.

Antoy, S. and Hanus, M. 2010. Functional logic programming. *Communications of the ACM* 53, 4, 74–85.

Antoy, S., Hanus, M., Jost, A. and Libby, S. 2020. ICurry. In *Declarative Programming and Knowledge Management - Conference on Declarative Programming (DECLARE 2019) 2020*, Springer LNCS, Berlin, Heidelberg, 12057, 286–307.

Antoy, S. and Jost, A. 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)* 2016, Springer LNCS, Berlin, Heidelberg, 10184, 97–113.

Baader, F. and Nipkow, T. 1998. *Term Rewriting and All That.* Cambridge University Press, Cambridge, UK.

Böhm, J., Hanus, M. and Teegen, F. 2021. From non-determinism to goroutines: A fair implementation of Curry in Go. In *Proceedings of the 23rd International Symposium on Principles and Practice of Declarative Programming (PPDP 2021) 2021*, ACM Press, New York, NY, USA, 16:1–16:15.

Braßel, B., Hanus, M. and Huch, F. 2004. Encapsulating non-determinism in functional logic computations. *Journal of Functional and Logic Programming* 2004, 6.

Braßel, B., Hanus, M., Peemöller, B. and Reck, F. 2011. KiCS2: A new compiler from Curry to Haskell. In *Proceedings of the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011) 2011*, Springer LNCS, Berlin, Heidelberg, 6816, 1–18.

Braßel, B. and Huch, F. 2007. On a tighter integration of functional and logic programming. In *Proceedings of the APLAS 2007*, Springer LNCS, Berlin, Heidelberg, 122–138.

Christiansen, J., Hanus, M., Reck, F. and Seidel, D. 2013. A semantics for weakly encapsulated search in functional logic programs. In *Proceedings of the 15th International Symposium on Principle and Practice of Declarative Programming (PPDP'13) 2013*, ACM Press, New York, NY, USA, 49–60.

de Moura, L. and Bjørner, N. 2008. Z3: An efficient SMT solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008) 2008*, Springer LNCS, 4963, 337-337–340-340.

Eisenberg, R. A., Vytiniotis, D., Jones, S. L. P. and Weirich, S. 2014. Closed type families with overlapping equations. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, S. Jagannathan and P. Sewell, Eds. ACM, San Diego, CA, USA, 671–684.

Fischer, S., Kiselyov, O. and Shan, C. 2011. Purely functional lazy nondeterministic programming. *Journal of Functional Programming* 21, 4-5, 413–465.

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

Hanus, M. 2011. Declarative processing of semistructured web data. In *Technical Communications of the 27th International Conference on Logic Programming 2011*, Vol. 11, Leibniz International Proceedings in Informatics (LIPIcs), 198–208.

Hanus, M. 2012. Improving lazy non-deterministic computations by demand analysis. In *Technical Communications of the 28th International Conference on Logic Programming 2012*, Vol. 17, Dagstuhl, Germany, Leibniz International Proceedings in Informatics (LIPIcs), 130–143.

Hanus, M. 2013. Functional logic programming: From theory to Curry. In *Programming Logics - Essays in Memory of Harald Ganzinger 2013*, Springer LNCS, Berlin, Heidelberg, 7797, 123–168.

Hanus, M. 2024. Improving logic programs by adding functions. In *Proceedings of the 34th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2024) 2024*, Springer LNCS, 14919, 27–44.

Hanus, M., Antoy, S., Braßel, B., Engelke, M., Höppner, K., Koj, J., Niederau, P., Sadre, R., Steiner, F. and Teegen, F. 2025. PAKCS: The Portland Aachen Kiel Curry System. https://www.curry-lang.org/pakcs/.

Hanus, M., Peemöller, B. and Reck, F. 2012. Search strategies for functional logic programming. In *Proceedings of the 5th Working Conference on Programming Languages (ATPS'12) 2012*, Springer LNI, Bonn, 199, pp. 61–74.

Hanus, M., Prott, K.-O. and Teegen, F. 2022. A monadic implementation of functional logic programs. In *Proceedings of the 24th International Symposium on Principles and Practice of Declarative Programming (PPDP 2022) 2022*, ACM Press, 1:1:–1:15.

Hanus, M. and Skrlac, F. 2014. A modular and generic analysis server system for functional logic programs. In *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation (PEPM'14) 2014*, ACM Press, 181–188.

Hanus, M. and Teegen, F. 2020. Adding Data to Curry. In *Declarative Programming and Knowledge Management - Conference on Declarative Programming (DECLARE 2019) 2020*, Springer LNCS, 12057, 230–246.

Hanus, M. and Teegen, F. 2021. Memoized pull-tabbing for functional logic programming. In *Proceedings of the 28th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2020) 2021*, Springer LNCS, Berlin, Heidelberg, 12560, 57–73.

Hanus, M. 2016. Curry: An integrated functional logic language (vers. 0.9.0). https://www.curry-lang.org.

https://doi.org/10.1017/S1471068426100453 Published online by Cambridge University Press

Huet, G. and Lévy, J.-J. 1991. Computations in orthogonal rewriting systems. In *Computational Logic: Essays in Honor of Alan Robinson*, J.-L. Lassez and G. Plotkin, Eds. Cambridge, Massachusetts, MIT Press, 395–443.

Hussmann, H. 1992. Nondeterministic algebraic specifications and nonconfluent term rewriting. *Journal of Logic Programming* 12, 237–255.

Johnsson, T. 1985. Lambda lifting: Transforming programs to recursive functions. In *Functional Programming Languages and Computer Architecture*, Springer LNCS, Berlin, Heidelberg, 201, 190–203.

Karachalias, G., Koprivec, F., Pretnar, M. and Schrijvers, T. 2021. Efficient compilation of algebraic effect handlers. *Proceedings of the ACM on Programming Languages* 5, 1–28. OOPSLA.

Lloyd, J. 1987. *Foundations of Logic Programming*, second, extended ed. Springer, Berlin, Heidelberg.

López-Fraguas, F. and Sánchez-Hernández, J. 1999. TOY: A multiparadigm declarative system. In *Proceedings of RTA'99 1999*, Springer LNCS, Berlin, Heidelberg, 1631, 244–247.

Moreno-Navarro, J., Kuchen, H., Loogen, R. and Rodríguez-Artalejo, M. 1990. Lazy narrowing in a graph machine. In *Proceedings of the Second International Conference on Algebraic and Logic Programming 1990*, Springer LNCS, 463, 298–317.

Perry, N. 2005. The implementation of practical functional programming languages. Ph.D. thesis, University of London.

Petricek, T. 2012. Evaluation strategies for monadic computations. *Electronic Proceedings in Theoretical Computer Science* 76, 68–89.

Peyton Jones, S. 2003. *Haskell 98 Language and Libraries—The Revised Report*, Cambridge, UK, Cambridge University Press.

Peyton Jones, S., Gordon, A. and Finne, S. 1996. Concurrent Haskell. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages (POPL'96) 1996*, ACM Press, 295–308.

Peyton Jones, S., Marlow, S. and Elliott, C. 2000. Stretching the storage manager: Weak pointers and stable names in Haskell. In *Implementation of Functional Languages 2000*, P. Koopman and C. Clack, Eds. Berlin, Heidelberg, Springer Berlin Heidelberg, 37–58.

Peyton Jones, S., Shields, M., Launchbury, J. and Tolmach, A. 1998. Bridging the gulf: A common intermediate language for ML and Haskell. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages 1998, POPL '98*, Association for Computing Machinery, New York, NY, USA, 49–61.

Prott, K., Teegen, F. and Christiansen, J. 2023. Embedding functional logic programming in Haskell via a compiler plugin. In *Practical Aspects of Declarative Languages - 25th International Symposium, PADL 2023, Boston, MA, USA, January 16-17, 2023, Proceedings 2023*, M. Hanus and D. Inclezan, Eds. Vol. 13880 of Lecture Notes in Computer Science, Springer, Boston, MA, USA, 37–55.

Reddy, U. 1985. Narrowing as the operational semantics of functional languages. In *Proceedings of the IEEE International Symposium on Logic Programming 1985*, IEEE Computer Society, Boston, 138–151.

Robinson, J. 1965. A machine-oriented logic based on the resolution principle. *Journal of the ACM* 12, 1, 23–41.

Slagle, J. 1974. Automated theorem-proving for theories with simplifiers, commutativity, and associativity. *Journal of the ACM* 21, 4, 622–642.

Teegen, F., Prott, K.-O. and Bunkenburg, N. 2021. Haskell^{-1}: Automatic function inversion in Haskell. In *Proceedings of the 14th ACM SIGPLAN International Symposium on Haskell*, Association for Computing Machinery, New York, NY, USA, 41–55.

https://doi.org/10.1017/S1471068426100453 Published online by Cambridge University Press

Wadler, P. 1985. How to replace failure by a list of successes: A method for exception handling, backtracking, and pattern matching in lazy functional languages. In *Conference on Functional Programming and Computer Architecture (FPCA'85) 1985*, Springer LNCS, Berlin, Heidelberg, 201, 113–128.

Wadler, P. 1990. Comprehending monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming 1990*, ACM, New York, NY, USA, 61–78.

Wadler, P. 1997. How to declare an imperative. *ACM Computing Surveys* 29, 3, 240–263.

https://doi.org/10.1017/S1471068426100453 Published online by Cambridge University Press