A Monadic Implementation of Functional Logic Programs

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

Kai-Oliver Prott
CAU Kiel
Institut für Informatik
Kiel, Germany
kpr@informatik.uni-kiel.de

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

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 to 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 efforts. 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. We demonstrate that our approach yields a promising performance that outperforms current compilers for Curry.

CCS CONCEPTS

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

KEYWORDS

Declarative programming, non-determinism, memoization, monads, implementation

ACM Reference Format:
Michael Hanus, Kai-Oliver Prott, and Finn Teegen. 2022. A Monadic Implementation of Functional Logic Programs. In *24th International Symposium on Principles and Practice of Declarative Programming (PPDP 2022), September 20–22, 2022, Tbilisi, Georgia.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3551357.3551370

1 INTRODUCTION

Declarative programming emphasizes the principle to express 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. 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 [39] uses these formulas to compute solutions to a given query.

Functional logic languages [12, 23] 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 [28]. Non-deterministic applications of program rules with overlapping left-hand sides ensure that solutions are computed whenever they exist [32]. The combination of these techniques is called *narrowing* [38, 40]. *Needed narrowing* is a demand-driven variant which is optimal w.r.t. the length of successful derivations and the number of computed solutions [4, 7].

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 (for source programs we use Curry syntax [27] which is close to Haskell [36]):[1]

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

`not` is a standard function whereas `aBool` is a *non-deterministic operation* [21] 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, e.g., `aBool` has values `False` and `True`. They are an important concept of contemporary functional logic languages. (see [12, 21] for more details). Non-deterministic operations can be used like any other operation, e.g., in data structures or as arguments to other operations, for example 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.

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

For instance, some implementations use *backtracking* to handle non-determinism, in particular, implementations of functional logic languages which compile into Prolog, like PAKCS [8, 24] or TOY [33]. Backtracking implements a choice by selecting one alternative to proceed 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 can be avoided by keeping all alternatives in one computation structure (a graph) and using a fair strategy to explore this structure.

Pull-tabbing is an approach to implement this idea. It was first sketched in [3] and formally explored in [6]. A pull-tab step is a local transformation that moves a choice in a (demanded) argument of an operation outside this operation. For instance,

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

is a pull-tab step. Pull-tabbing is used in implementations targeting complete search strategies, e.g., KiCS [18], KiCS2 [17], or Sprite [14]. 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 to duplication of computations. The latter is a serious problem of pull-tabbing implementations [22]. For instance, consider the additional operations

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

Then 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 values `True` are unintended. 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 [6]. However, this does not fix the duplication of choices, which can be detrimental to performance due to repeated evaluation.

The Curry compiler KiCS2 [17] 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. Hence, an optimization is proposed in [22] to eagerly evaluate 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) [26], has been used in Curry implementations which transform source programs into Julia programs [26] or Go programs [16].

Since MPT has been implemented only in imperative target languages, it was stated as a question whether the "MPT scheme can be combined with the purely functional implementation approach of KiCS2" [26]. 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 [16, 17].

In this paper we present 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 of non-deterministic branches and also advanced features of contemporary functional logic languages, such as functional patterns [9] and set functions to encapsulate non-determinism [11].

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 this paper, our implementation is the first Curry implementation that integrates all features of Curry with a promising performance.

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 an explicit representation of free (logic) variables. We evaluate our approach in Section 7 before we discuss related work in Section 8 and conclude with some future work in Section 9.

2 FUNCTIONAL LOGIC PROGRAMMING

We assume familiarity with functional programming and Haskell so that 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 [12, 23] and in the language report [27].

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] [36] but allows *free* (*logic*) *variables* in conditions and right-hand sides of defining rules. In contrast to Haskell, rule selection is non-deterministic, i.e., if more than one rule is applicable, all applicable rules are tried. The operational semantics is based on an optimal evaluation strategy [4, 7]—a conservative extension of lazy functional programming and logic programming.

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`**):

```
                                                          Curry
```

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

```
last xs | ys ++ [x] =:= xs = x
  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* [21] are an important feature of contemporary functional logic languages. They are conceptually equivalent to free variables (as shown in [10]) and can be nested as other functions. This is due to the fact that non-deterministic operations return (non-deterministically) individual values rather than sets of values (although their declarative meaning can be specified with a set-valued semantics [21]). 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 [42], 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 considerable smaller search spaces (see [12, 21] for more detailed discussions).

As mentioned in Section 1, the occurrence of non-deterministic operations as arguments might cause a semantical ambiguity when such arguments are used multiple times. For instance, if we evaluate the expression `xorSelf aBool` as in standard term rewriting [15], i.e., 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
```

The result `True` is unintended for the operation `xorSelf`. 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 evaluation strategy and exclude such unintended results, González-Moreno et al. [21] 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 [29], 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 [28], we cannot directly map functional logic programs into Haskell programs due to the non-deterministic features. Thus, a correct mapping requires to model non-determinism *and* sharing 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 [44], *set functions* [11] are useful to encapsulate search in a evaluation-independent manner, and *functional patterns* [9] 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

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 [42]: 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, e.g., 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
```

`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 [42] where `return` creates a singleton list, `mzero` an empty list, and `mplus` concatenates the argument list. Then the monadic representation of `aBool` returns the list of its two values:

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

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

$$\llbracket \forall\alpha_1 \ldots \alpha_n .\ \phi \Rightarrow \tau \rrbracket^t \coloneqq \forall\alpha_1 \ldots \alpha_n .\ \llbracket \phi \rrbracket^t \Rightarrow \texttt{Curry}\ (\llbracket \tau \rrbracket^i) \qquad \text{(Polymorphic type)}$$

$$\llbracket \langle \kappa_1\ \tau_1,\ \ldots,\ \kappa_n\ \tau_n \rangle \rrbracket^t \coloneqq \langle \texttt{rename}(\kappa_1) \llbracket \tau_1 \rrbracket^i,\ \ldots,\ \texttt{rename}(\kappa_1) \llbracket \tau_1 \rrbracket^i \rangle \qquad \text{(Context)}$$

$$\llbracket \tau_1 \to \tau_2 \rrbracket^i \coloneqq \llbracket \tau_1 \rrbracket^i \to_{\texttt{C}} \llbracket \tau_2 \rrbracket^i \qquad \text{(Function type)}$$

$$\llbracket \tau_1\ \tau_2 \rrbracket^i \coloneqq \llbracket \tau_1 \rrbracket^i\ \llbracket \tau_2 \rrbracket^i \qquad \text{(Type application)}$$

$$\llbracket \chi \rrbracket^i \coloneqq \texttt{rename}(\chi) \qquad \text{(Type constructor)}$$

$$\llbracket \alpha \rrbracket^i \coloneqq \alpha \qquad \text{(Type variable)}$$

Figure 1: Type lifting $\llbracket \circ \rrbracket^t$

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, e.g., 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 [43] 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 a 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. Thus, we explicitly model sharing using an approach adapted from both Fischer et al. [20] and Petricek [35]. 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 lifted functions to increase readability.

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

Types. On the type level, the monadic lifting replaces type constructors with their effectful counterparts and wraps the result and argument of each function type in `Curry`. However, any quantifiers and 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 lifting 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                                      Curry
```

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 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 in-lining and rewrite optimizations can get rid of some of the overhead introduced here. Such optimizations are possible with the Glasgow Haskell Compiler that we target.

$$\begin{aligned}
[\![\texttt{data}\ D\ \alpha_1 \ldots \alpha_n\ \texttt{=}\ C_1\ \texttt{|}\ \ldots\ \texttt{|}\ C_n\ ]\!]^d &\coloneqq \texttt{data}\ \texttt{rename}(D)\ \alpha_1 \ldots \alpha_n\ \texttt{=}\ [\![C_1]\!]^c\ \texttt{|}\ \ldots\ \texttt{|}\ [\![C_n]\!]^c && \text{(Data type)}\\
[\![C\ \tau_1 \ldots \tau_n]\!]^c &\coloneqq \texttt{rename}(C)\ [\![\tau_1]\!]^t \ldots [\![\tau_n]\!]^t && \text{(Constructor)}
\end{aligned}$$

Figure 2: Data type lifting $[\![\circ]\!]^d$

$$\begin{aligned}
[\![x]\!]^e &\coloneqq x && \text{(Variable)}\\
[\![\lambda x \to e]\!]^e &\coloneqq \texttt{return (Func (}\lambda\texttt{y} \to alias(y, x, [\![e]\!]^e)\texttt{))} && \text{(Abstraction)}\\
[\![e_1\ e_2]\!]^e &\coloneqq [\![e_1]\!]^e \gg\!= \lambda\texttt{(Func}\ y\texttt{)} \to y\ [\![e_2]\!]^e && \text{(Application)}\\
[\![C]\!]^e &\coloneqq \texttt{return (Func (}\lambda y_1 \to \ldots\ \texttt{return (Func (}\lambda y_n \to \texttt{return (rename(}C\texttt{)}\ y_1 \ldots y_n\texttt{)))}\ldots\texttt{))} && \text{(Constructor)}\\
[\![\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2]\!]^e &\coloneqq \texttt{let}\ y\ \texttt{=}\ [\![e_1]\!]^e\ \texttt{in}\ alias(y, x, e_2) && \text{(Let Expression)}\\
[\![\texttt{(?)}]\!]^e &\coloneqq \texttt{return (Func (}\lambda y_1 \to \texttt{return (Func (}\lambda y_2 \to\ y_1\ \texttt{`mplus`}\ y_2\texttt{))))} && \text{(Choice operator)}\\
[\![\texttt{failed}]\!]^e &\coloneqq \texttt{mzero} && \text{(Failed computation)}\\
[\![\texttt{case}\ e\ \texttt{of}\ \{br_1;\ldots;br_n\}]\!]^e &\coloneqq [\![e]\!]^e \gg\!= \lambda y \to \texttt{case}\ y\ \texttt{of}\ \{[\![br_1]\!]^b;\ldots;[\![br_n]\!]^b\} && \text{(Case Expression)}\\
[\![C\ x_1 \ldots x_n \to e]\!]^b &\coloneqq \texttt{rename}(C)\ \ y_1 \ldots y_n \to alias(y_1, x_1, (\ldots(alias(y_n, x_n, [\![e]\!]^e)))) && \text{(Case Branch)}\\
\\
alias(x_{new}, x_{old}, e) &\coloneqq \texttt{share}\ x_{new} \gg\!= \lambda x_{old} \to e && \text{(Aliasing)}
\end{aligned}$$

Figure 3: Expression lifting $[\![\circ]\!]^e$ (where $y, y_1, \ldots, y_n$ are fresh variables)

Data Type Declarations. As discussed in [20] in detail, the arguments of data constructors need also be transformed into monadic values in order to achieve non-strictness of data constructors. Thus, we need to modify data type definitions (except for primitive data types). 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 non-strict data constructors, we lift every constructor. As the (partial or full) application of a constructor can never be non-deterministic by itself, we neither have to lift the result type of the constructor nor wrap the function arrow. This allows us to lift 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 lifting 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))
```

Because the lifting of a constructor differs from the lifting 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.

Functions. The type of a lifted function serves as a guide for the lifting of functions and expressions. We can derive three rules for our lifting of expressions from our lifting 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 an implication, 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 [30]), 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 `(?)`. Actually, any functional logic program or constructor-based conditional term rewriting system can be transformed into this flat form [5]. For instance, the operation `insert` defined in Section 2 is desugared into the following flat form.

```
                                                    Curry
```

[5]The flat form of programs is also used for the semantics [2] and implementation [13] of functional logic programs.

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

Figure 3 presents the rules of our monadic transformation. 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.

Lambda Abstractions. A lambda abstraction is translated by wrapping it in a `return` as well as the lifted function constructor `Func` and translating the inner expression. Additionally, we apply `share` to the argument of the lambda abstraction. This is necessary to avoid the re-evaluation of this argument when it occurs multiple times in the body or is duplicated by other functions. Evaluating arguments at most once is important in a call-by-need semantics to avoid superfluous computations and is semantically relevant in a non-deterministic setting to ensure the call-time choice semantics (see Section 2).

Applications. We extract the "real" function from the monad before applying it in the lifted setting. An application of a function to more than one argument is represented as multiple nested applications. While Petricek [35] uses `share` in the application of a function to an argument, we use `share` whenever a variable is brought into scope. Because we consider data types and case expressions in contrast to Petricek, variables are brought into scope in lambda, let, and case expressions.

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 (we slightly modify the target code to improve its readability)

```
notC = return ∘ Func $ λy →                          Haskell
  share y ≫= λx → x ≫= λz →
  case z of FalseC → return TrueC
            TrueC  → return FalseC
```

As a further example with a more complex pattern matching, consider the transformed variant of the `perm` operation from Section 2.

```
permC :: Curry (ListC a →C ListC a)                  Haskell
permC = return ∘ Func $ λarg' →
  share arg' ≫= λarg → arg ≫= λxs → case xs of
    NilC         → return NilC
    ConsC y' ys' →
      share y' ≫= λy → share ys' ≫= λys →
        insertC ≫= λi1 →
          i1 y ≫= λi2 → i2 (permC ≫= λp → p ys)
```

Although the code got significantly more complex, the user will not have to read or write such code since the transformation can be fully automated.

4 A HISTORY OF MONADIC PULL-TABBING

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

4.1 Tree-based Non-determinism

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)
```

Thus, `Leaf` represents a single value, `Empty` no value, i.e., 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 so that we omit them here (see Appendix A).

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 [17] which has options to select various search strategies (e.g., depth-/breadth-first, iterative deepening, parallel) [25].[6]

4.2 Fingerprinting

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, i.e., 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, i.e., selects the same (left/right) branch for identically decorated choices, then the computed values are the intended ones [6]. Therefore, implementations of functional logic languages which use pull-tabbing to support flexible and operationally complete search

[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.

strategies [14, 17] 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 [17].

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 [17] but it might be much slower than other Curry implementations for particular uses of non-deterministic operations [22].

4.3 Explicit Sharing of Computations

A solution to pull-tabbing without fingerprinting is proposed by Fischer et al. [20]. 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.

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 of implementations shown in the previous section. It is based on a solution to pull-tabbing without fingerprinting that enables sharing across non-determinism, proposed in [26], but here the results of non-deterministic computations are memoized in a different way so that it fits into the functional monadic transformation.

An implementation based on the monadic transformation presented in the previous section has to implement the functional interface on which this transformation is based, i.e., an appropriate monad instance and an implementation of the operation `share`. This is quite similar to the approach of Fischer et al. [20] 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 functional 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, 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 *memoized pull-tabbing* (MPT) [26] and formally described in [16], avoids the aforementioned problem of pull-tabbing. Its efficiency can compete with backtracking-based implementations [26]. Furthermore, it is a good basis for operationally complete implementations of functional logic languages. Existing implementations [16, 26] use imperative target languages. In the following, we discuss a high-level Haskell-based implementation which 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
```

In the next step, we define a state where we can 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, by using the `freshID` below.

```
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 Glasgow Haskell Compiler (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
  mempty = 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 :: Shareable a                                   Haskell
      ⇒ Curry a → Curry (Curry a)
```

Compared to the type described in Section 3, we have added the type class constraint `Shareable`. This constraint was introduced in a more generalized form by Fischer et al. [20]. The class contains a function `shareArgs`, which essentially calls `share` on each monadic component of a lifted data type so that even the results of lazy computations deep inside a data structure are shared and memoized. For example, the `Shareable` instance for the lifted list type can be defined as follows.

```
instance Shareable a                                   Haskell
  ⇒ Shareable (ListC a) where
    shareArgs NilC         = return NilC
    shareArgs (ConsC x xs) =
      ConsC <$> share x <*> share xs
```

Continuing with the implementation of `share`, we use `shareArgs` for the purpose explained above and memoize the possible computation of the argument with a new operation `memo`.

```
share ma = memo (ma ≫= shareArgs)                      Haskell
```

The function `memo` 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 `memo` step by step.

```
memo :: Curry a → Curry (Curry a)                      Haskell
memo (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` 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 `memo` 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 right 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                Haskell
                 → Set ID → Maybe a
lookupTaskResult trRef i p =
    msum (map (lookupHeap 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 `Maybe a` results using the monoidal `msum`, we obtain the first result if it exists, or `Nothing`

[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.

otherwise. Note that there can only ever be at most one valid result in the map.

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

```haskell
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 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 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 `memo`. 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.

```haskell
Just res → return res
```

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.

```curry
notIf :: Bool → Bool
notIf x = let f = not x
          in if x then f else f
```

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`.

```haskell
notIfC :: Curry (BoolC →C BoolC)
notIfC = return ∘ Func $ λx' → share x' ≫= λx →
  let f' = notC ≫= λn → n x
  in share f' ≫= λf → x ≫= λx' → case x' of
       TrueC  → f
       FalseC → f
```

We see that `x` is shared before `f` is introduced and `x` is evaluated before `f` since it is demanded by `notC`. When the computation of `f` 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 `f` is deterministic so that the value of `f` is memoized globally as `TrueC`. In consequence, our implementation fails in situations where a shared value (`f`) 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 introduced non-determinism. Therefore, we need to adapt the implementation. The relevant changes are in both **case**-branches.

```haskell
case lookupTaskResult trRef b2 p2 of
  Just (wasND, res) →
    | wasND → do
        i ← freshID
        let newParents = Set.insert b2 p2
        put (MemoState i newParents idSupply2)
        return res
    | otherwise →
        return res
  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
```

5.4 Improving Performance of Sharing

There is one major performance bottleneck with this implementation. The problem is that `share` traverses its argument recursively via `shareArgs`, even when a previous `share` has done so already. Repeated sharing of a computation is unneccessary 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 two ideas.

(1) We prevent any unneccessary insertions of `share` during monadic lifting. If a value is used at most once in its scope, a `share` is not required. Consider the following example of a function that reverses the order of elements in a list.

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

Here, both `x` and `xs` are used exactly once in the second rule. Thus, we can avoid sharing them explicitly. In the rules given in Figure 3, we replace the *alias* rule as seen in Figure 4. Note that, since Curry is a lazy language, we can only over-approximate if a value will be used at least twice. For example, in `const x x`, we will still consider `x` to be shared, even though `const` does not evaluate its first argument.

$$alias(x_{new}, x_{old}, e) \coloneqq \begin{cases} \texttt{share } x_{new} \gg\!= \lambda x_{old} \to \texttt{e} & \text{if } x_{old} \text{ might be used at least twice in } e \\ \texttt{let } x_{old} = x_{new} \texttt{ in e} & \text{if } x_{old} \text{ is guaranteed to be used at most once in } e \end{cases} \quad \text{(Aliasing)}$$

Figure 4: Improved *alias* rule

(2) Additionally, we tag values with information on whether or not they have been shared already. If a value is already shared, we can skip recursing into its data structure with `shareArgs`. This greatly improves performance for recursive functions over inductive data types. The following example of a contrived `reverse` function illustrates this.

```
reverse xs = if null xs                          Curry
  then xs
  else reverse (tail xs) ++ [head xs]
```

Here, `xs` is used at least twice in each branch. Thus, sharing it is necessary to achieve the desired semantics explained in Section 2. However, we will share the whole list in the first call to `reverse`. Any `share` in a subsequent recursive call is unnecessary. By tagging shared values and not sharing tagged values again we can avoid some work. Since this optimization is not relevant to this paper, we omit its code. It is available in the Github repository for this project.[8]

6 EXTENDING THE MONAD

An important advantage of our monadic implementation is that one can implement new features by modifying the monad without modifying the translation of source programs. While we do not have the space to discuss every extension we implemented, we want to show at least the implementation of one extension as an example. Therefore, we will demonstrate the extension of our monad by free (logic) variables in order to support unification with variable bindings instead of instantiating free variables to all possible values.

Free variables are an important feature of logic programming. They denote unknown values which are fixed during the computation. It is well known that free variables can be replaced by value generators, i.e., non-deterministic operations that yield the (possibly infinite) set of values of a given type [10]. Although this works from a declarative point of view, a crucial feature of logic programming is *unification* to compute with partial information, i.e., 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 versa) without instantiating them to a value. In contrast, if `x` and `y` are generators for an infinite set of values, their evaluation leads to an infinite search space. In order to support an efficient implementation of unification, we need to explicitly represent free variables so that we can implement bindings without value generation.

For this purpose, we use the approach taken by Teegen et al. [41] 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, i.e., it is partially instantiated. Thus, variables only get narrowed on demand and a (≫=) is only ever used to extract the scrutinee of a **`case`**-match or to extract a function to be applied.[9] In both cases, we need an explicit value instead of a free variable in order to continue the evaluation.

Teegen et al. define a type class `Narrowable`, which enumerates all constructors of the given type and fills their arguments with new free variables.[10]

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

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

```
instance Narrowable a ⇒ Narrowable (ListC a)    Haskell
 where
  narrow [NilC, ConsC free free]
```

Any free variables are created using the `free` function that will be introduced later. Next, we also need to extend our state with a heap to store the bindings of free variables that have been narrowed already. Not all free variables have the same type, thus, our heap has to be untyped using existential data types [34] and `unsafeCoerce` as seen below. Using unsafe features here will not go wrong at run time, since they are captured behind a safe interface.

```
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 give the new type for our memoization monad extended with free variables. Apart from the state, it differs from our previous monad by having an `FLVal a` instead of a plain `a` in its result type. This new type differentiates between values 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. Note that we cannot derive required instances (e.g., `Monad`) automatically anymore.

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

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

[8]Available at: https://github.com/cau-placc/curry-monad

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

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

In the following we quickly show the implementation of `free` that we mentioned earlier. It uses the new `FLVal` type and provides a fresh identifier and the required `Narrowable` constraint. 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 give the new `Monad` instance for the `CurryFree` type, we introduce another helper function for the instantiation of free variables. This instantiation starts by generating a lifted 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 indentifiers to monadic computations that have been shared already. We then combine the computations for each constructor by non-deterministic choices using `msum`. It is imperative that we write a *shared* variant of the corresponding value to the heap. 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.

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

Finally, we give the implementation of the `Monad` instance. The only interesting bit is that we case match on the result of the first argument so that we can check whether it is a variable or a value. Recall that we want to instantiate variables when they are pattern matched on, which corresponds to a (≫=) in the monadic translation. In case the result is a variable that is yet unbound, we first instantiate that variable before we continue with the computation. Otherwise, we can apply the continuation from the second argument.

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

Using this monad with an explicit representation of free variables, it is possible to implement a standard unification procedure by inserting variable bindings directly onto the heap rather than instantiating these variables. An implementation of a Prolog-like unification algorithm based on this representation is given in Appendix B.

As mentioned in Section 2, a few other extensions for high-level declarative programming have been proposed, like functional patterns [9], which support deep pattern matching, or set functions [11] to encapsulate non-deterministic operations. Due to lack of space, we have to omit their implementations as well. The implementation of these features are adapted from the papers cited above, but there are some unique challenges with respect to our memoization for each of them. The full implementation is available in our Github repository.

7 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 allow for sharing of deterministic values even across non-determinism.

Due to the monadic abstraction, the implementation of our prototype compiler allowed for a clean separation between the runtime system (i.e., the monad and `share` implementation) and the monadic transformation. Although the full compiler is still under implementation, the current implementation shows that the code size is smaller (thus, better maintainable) than KiCS2, which also compiles into Haskell.

In order to evaluate the overall efficiency of our high-level implementation, we compare it on various benchmark programs[11] with three other major Curry implementations. PAKCS [24], which is part of Debian and Ubuntu Linux distributions, compiles to Prolog (SWI-Prolog 8.4.2) so that its search strategy is based on backtracking. KICS2 [17] compiles to Haskell (GHC 8.4) and implements non-determinism by pure pull-tabbing without memoization. Curry2Go [16] compiles to Go (1.16.1) and uses an imperative implementation of memoization. All benchmarks were executed on a Linux machine running Debian 11 with an Intel Core i7-7700K (4.2GHz) 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. Since a complete compiler environment including an interactive REPL (read-eval-print loop) to evaluate arbitrary expressions based on our monadic implementation is under construction, we manually used the GHC to compile the generated monadic code and to measure the run times shown below.

Figure 5 shows the timings for various programs. The first four of these test the properties that our implementation was designed for, while the last two benchmarks are purely deterministic programs. Times for our approach are both "Monadic MPT" colums, where the latter one integrates an optimization for fully deterministic sub-computations. We will focus on the non-optimized version of "Monadic MPT" first and talk about the optimized implementation at the end of this section. The benchmarks are based on the ones used by Böhm et al. [16] to evaluate their implementation of memoized pull-tabbing.

[11]The benchnmarks are available in our Github repository.

Program	Monadic MPT	Monadic MPT + det. optim.	KiCS2	PAKCS	Curry2Go
AddNum10	0.29	0.29	13.70	0.41	0.68
AddNum5	0.19	0.19	4.36	0.33	0.43
YesSharingND	0.62	0.20	0.42	33.65	4.62
NoSharingND	1.33	0.21	0.78	34.07	2.97
PermSort	2.46	2.44	2.80	10.30	5.96
SortPrimes	1.25	0.01	0.03	98.56	1.02
NaiveReverse	0.24	0.09	0.20	6.46	1.15
Queens10	2.67	0.03	0.45	185.31	27.41

Figure 5: Timings (in seconds) of various programs evaluated with different compilers, green marks best time

The first benchmark checks memoization, i.e., the avoidance of repeated pull-tabbing. The contrived program to test this property gets some number `n` as input and non-deterministically generates a number `x` between `0` and `n` (inclusive). Afterwards, `x` is added ten times to itself.

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

addNum10 :: Int → Int
addNum10 n = let x = someNum n
             in x+x+x+x+x+x+x+x+x+x+x
```

With memoized pull-tabbing, the choice for `x` should be made only once and not on each addition of `x`. Looking at the times in Figure 5, our implementation `AddNum10` takes less than double the time of a smaller test `AddNum5` where `x` is added only five times to itself. This is in contrast to KiCS2 which does not use memoization to implement pull-tabbing.

The next benchmark uses the `primes` example from Section 4 to create an expensive deterministic computation that yields the 800th prime number.

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

We can use this expensive computation 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.[12]

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

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

And indeed, the second test takes approximately twice as long as the first test. Thus, we can conclude that our implementation shares deterministic values across non-determinism.

The final benchmarks are taken from [26]. `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. `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 both PAKCS and 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` where avoiding repeated pull-tabbing is relevant, we are expectedly faster than KiCS2 and even faster than Curry2Go. PAKCS is about as fast, since it uses backtracking with Prolog and not pull-tabbing.

Unfortunately, the non-optimized version is a bit slower than Curry2Go and much slower than KiCS2 in the mixed benchmark `SortPrimes`. While this seems surprising at first, the speed of KiCS2 stems from the fact that it 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.

While we are not confident that we can replicate exactly the same optimization from KiCS2 for our monadic implementation, we have tested an experimental approach to optimize our monadic implementation for purely deterministic computations that are only applied to deterministic values. For such programs, we can generate non-monadic, purely functional Haskell code that is much faster. This requires a non-determinism analysis just like KICS2 uses. The results in the column for our optimized approach in Figure 5 show that our implementation is the fastest implementation for these benchmarks, although one can still construct examples where KiCS2 could be faster.

8 RELATED WORK

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

8.1 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.

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

The compiler most similar to our work is KiCS2 [17]. 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 in-lined by augmenting every data type with a constuctor 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 provides for 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 [19].

Other related compilers are Curry2Go [16] and its Julia-based predecessor [26]. 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 [14] 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 yet available. The results published in [14] indicate that the performance of Sprite behaves similarly to that of KiCS2.

The PAKCS compiler from Curry to Prolog [24] 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.

8.2 Monadic Intermediate Languages

Our transformation basically models the denotational semantics of Curry explicitly. Peyton Jones et al. [37] 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.

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. [1] 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 [31]. However, Eff is an entirely different language than Curry with different challenges.

9 CONCLUSIONS AND FUTURE WORK

In this work we developed an implementation of Curry which supports various advanced features of functional logic languages introduced during the last 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 have integrated other extensions in the code on our Github page. Due to the monadic structure of the target programs, these extensions can be implemented by modifying the monad only without changing the compilation scheme. Although the implementation of a complete Curry system based on our monadic scheme requires more work, the first benchmark results indicate a good performance compared to other compilers.

For future work, we will implement our idea on how to speed up purely deterministic sub-computations of a Curry program by integrating a compile-time analysis of deterministic operations. We will also combine the monadic lifting with our monadic model of Curry to implement a complete compiler for Curry.

ACKNOWLEDGMENTS

The authors are grateful to the anonymous reviewers for their helpful comments to improve this paper.

REFERENCES

[1] Andreas Abel, Marcin Benke, Ana Bove, John Hughes, and Ulf Norell. 2005. Verifying Haskell Programs Using Constructive Type Theory. In *Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell*. ACM Press, New York, NY, USA, 62–73. https://doi.org/10.1145/1088348.1088355

[2] E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. 2005. Operational Semantics for Declarative Multi-Paradigm Languages. *Journal of Symbolic Computation* 40, 1 (2005), 795–829. https://doi.org/10.1016/j.jsc.2004.01.001

[3] A. Alqaddoumi, S. Antoy, S. Fischer, and F. Reck. 2010. The Pull-Tab Transformation. In *Proc. of the Third International Workshop on Graph Computation Models*. Published Online, Enschede, The Netherlands, 127–132. Available at http://gcm2010.imag.fr/pages/gcm2010-preproceedings.pdf.

[4] S. Antoy. 1997. Optimal Non-Deterministic Functional Logic Computations. In *Proc. International Conference on Algebraic and Logic Programming (ALP'97)*. Springer LNCS 1298, Berlin, Heidelberg, 16–30. https://doi.org/10.1007/BFb0027000

[5] S. Antoy. 2001. Constructor-based Conditional Narrowing. In *Proc. of the 3rd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2001)*. ACM Press, New York, NY, USA, 199–206.

[6] S. Antoy. 2011. On the Correctness of Pull-Tabbing. *Theory and Practice of Logic Programming* 11, 4-5 (2011), 713–730. https://doi.org/10.1017/S1471068411000263

[7] S. Antoy, R. Echahed, and M. Hanus. 2000. A Needed Narrowing Strategy . *J. ACM* 47, 4 (2000), 776–822. https://doi.org/10.1145/347476.347484

[8] S. Antoy and M. Hanus. 2000. Compiling Multi-Paradigm Declarative Programs into Prolog. In *Proc. International Workshop on Frontiers of Combining Systems (FroCoS'2000)*. Springer LNCS 1794, Berlin, Heidelberg, 171–185. https://doi.org/10.1007/10720084_12

[9] S. Antoy and M. Hanus. 2005. Declarative Programming with Function Patterns. In *Proceedings of the International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'05)*. Springer LNCS 3901, Berlin, Heidelberg, 6–22. https://doi.org/10.1007/11680093_2

[10] S. Antoy and M. Hanus. 2006. Overlapping Rules and Logic Variables in Functional Logic Programs. In *Proceedings of the 22nd International Conference on Logic Programming (ICLP 2006)*. Springer LNCS 4079, Berlin, Heidelberg, 87–101. https://doi.org/10.1007/11799573_9

[11] S. Antoy and M. Hanus. 2009. Set Functions for Functional Logic Programming. In *Proceedings of the 11th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'09)*. ACM Press, New York, NY, USA, 73–82. https://doi.org/10.1145/1599410.1599420

[12] S. Antoy and M. Hanus. 2010. Functional Logic Programming. *Commun. ACM* 53, 4 (2010), 74–85. https://doi.org/10.1145/1721654.1721675

[13] S. Antoy, M. Hanus, A. Jost, and S. Libby. 2020. ICurry. In *Declarative Programming and Knowledge Management - Conference on Declarative Programming (DECLARE 2019)*. Springer LNCS 12057, Berlin, Heidelberg, 286–307. https://doi.org/10.1007/978-3-030-46714-2_18

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

[15] F. Baader and T. Nipkow. 1998. *Term Rewriting and All That*. Cambridge University Press, Cambridge, UK. https://doi.org/10.1017/CBO9781139172752

[16] J. Böhm, M. Hanus, and F. Teegen. 2021. From Non-determinism to Goroutines: A Fair Implementation of Curry in Go. In *Proc. of the 23rd International Symposium on Principles and Practice of Declarative Programming (PPDP 2021)*. ACM Press, New York, NY, USA, 16:1–16:15. https://doi.org/10.1145/3479394.3479411

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

[18] B. Braßel and F. Huch. 2007. On a Tighter Integration of Functional and Logic Programming. In *Proc. APLAS 2007*. Springer LNCS 4807, Berlin, Heidelberg, 122–138. https://doi.org/10.1007/978-3-540-76637-7_9

[19] J. Christiansen, M. Hanus, F. Reck, and D. Seidel. 2013. A Semantics for Weakly Encapsulated Search in Functional Logic Programs. In *Proc. of the 15th International Symposium on Principle and Practice of Declarative Programming (PPDP'13)*. ACM Press, New York, NY, USA, 49–60. https://doi.org/10.1145/2505879.2505896

[20] S. Fischer, O. Kiselyov, and C. Shan. 2011. Purely functional lazy nondeterministic programming. *Journal of Functional programming* 21, 4&5 (2011), 413–465. https://doi.org/10.1017/S0956796811000189

[21] J.C. González-Moreno, M.T. Hortalá-González, F.J. López-Fraguas, and M. Rodríguez-Artalejo. 1999. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming* 40 (1999), 47–87. https://doi.org/10.1016/s0743-1066(98)10029-8

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

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

[24] M. Hanus, S. Antoy, B. Braßel, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, F. Steiner, and F. Teegen. 2021. PAKCS: The Portland Aachen Kiel Curry System. Available at http://www.informatik.uni-kiel.de/~pakcs/.

[25] M. Hanus, B. Peemöller, and F. Reck. 2012. Search Strategies for Functional Logic Programming. In *Proc. of the 5th Working Conference on Programming Languages (ATPS'12)*. Springer LNI 199, Bonn, 61–74. https://doi.org/20.500.12116/18376

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

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

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

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

[30] T. Johnsson. 1985. Lambda Lifting: Transforming Programs to Recursive Functions. In *Functional Programming Languages and Computer Architecture*. Springer LNCS 201, Berlin, Heidelberg, 190–203. https://doi.org/10.1007/3-540-15975-4_37

[31] Georgios Karachalias, Filip Koprivec, Matija Pretnar, and Tom Schrijvers. 2021. Efficient Compilation of Algebraic Effect Handlers. *Proc. ACM Program. Lang.* 5, OOPSLA (Oct. 2021), 28. https://doi.org/10.1145/3485479

[32] J.W. Lloyd. 1987. *Foundations of Logic Programming*. Springer, second, extended edition, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-96826-6

[33] F. López-Fraguas and J. Sánchez-Hernández. 1999. TOY: A Multiparadigm Declarative System. In *Proc. of RTA'99*. Springer LNCS 1631, Berlin, Heidelberg, 244–247. https://doi.org/10.1007/3-540-48685-2_19

[34] Nigel Perry. 2005. *The Implementation of Practical Functional Programming Languages*. Ph.D. Dissertation. University of London.

[35] Tomas Petricek. 2012. Evaluation Strategies for Monadic Computations. *Electronic Proceedings in Theoretical Computer Science* 76 (2012), 68–89. https://doi.org/10.4204/EPTCS.76.7

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

[37] Simon Peyton Jones, Mark Shields, John Launchbury, and Andrew Tolmach. 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 (POPL '98)*. Association for Computing Machinery, New York, NY, USA, 49–61. https://doi.org/10.1145/268946.268951

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

[39] J.A. Robinson. 1965. A Machine-Oriented Logic Based on the Resolution Principle. *J. ACM* 12, 1 (1965), 23–41. https://doi.org/10.1145/321250.321253

[40] J.R. Slagle. 1974. Automated Theorem-Proving for Theories with Simplifiers, Commutativity, and Associativity. *J. ACM* 21, 4 (1974), 622–642. https://doi.org/10.1145/321850.321859

[41] Finn Teegen, Kai-Oliver Prott, and Niels Bunkenburg. 2021. Haskell^{-1}: Automatic Function Inversion in Haskell. In *Proceedings of the 14th ACM SIGPLAN International Symposium on Haskell (Haskell 2021)*. Association for Computing Machinery, New York, NY, USA, 41–55. https://doi.org/10.1145/3471874.3472982

[42] P. Wadler. 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)*. Springer LNCS 201, Berlin, Heidelberg, 113–128. https://doi.org/10.1007/3-540-15975-4_33

[43] P. Wadler. 1990. Comprehending Monads. In *Proc. 1990 ACM Conference on LISP and Functional Programming*. ACM, New York, NY, USA, 61–78. https://doi.org/10.1145/91556.91592

[44] P. Wadler. 1997. How to Declare an Imperative. *Comput. Surveys* 29, 3 (1997), 240–263. https://doi.org/10.1145/262009.262011

A MONAD INSTANCES FOR BINARY TREES

The Monad and MonadPlus instances for binary trees as introduced in Section 4.1 can be defined 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
```

B MONADIC UNIFICATION

The following code shows an implementation of a unification procedure based on the monad with an explicit representation of free variables introduced in Section 6. The important issue is the distinction between free variables and values (see type FLVal defined in Section 6). 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
      FreeState b ps h suppl ← get
      let h' = insertHeap i (CF (Var j)) h
      put (FreeState b ps h' suppl)
    (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 (Val i)
        Val x → return (Val x)
```

The instances of Unifiable can be schematically generated together with the lifting 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 unify the arguments of identical data constructors, as shown for lists.

```
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 seen in the Section 1 can be implemented as follows.

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