A compiler for Curry based on a GHC plugin

Fredrik Wieczerkowski

Bachelor's Thesis
September 2021

Programming Languages and Compiler Construction
Department of Computer Science
Kiel University

Advised by
Prof. Dr. Michael Hanus
M.Sc. Kai Prott
M.Sc. Finn Teegen

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel,

Abstract

The Curry programming language combines many features from functional and logic programming, yet most Curry compilers traditionally had to be written from scratch, despite the language's similarity to Haskell. Recent work by Prott explores an alternative approach that extends the Glasgow Haskell Compiler (GHC) with Curry's nondeterminism through a plugin. While ambient nondeterminism is at the heart of the Curry language, useful applications often arise from related concepts, many of which can be derived given only nondeterminism as a primitive.

Building on the plugin's nondeterminism, we therefore implement a fully-featured Curry system with support for free variables, unification, Input/Output (IO) and encapsulated search in this thesis, all while leveraging large parts of the existing GHC infrastructure, including language-level metaprogramming facilities, such as GHC Generics or Template Haskell. Additionally, we provide a Read-Eval-Print-Loop (REPL) for nondeterministic expressions as a convenient and interactive interface to this new Curry system.

Acknowledgements

I want to thank my advisors, Kai Prott, Finn Teegen and Prof. Dr. Michael Hanus, for making this project possible and for the helpful advice, insights and feedback both during the project and the writing phase. Special thanks also to my family and friends, for reading and commenting on various parts of this thesis.

Contents

List of Figures

Chapter 1

Introduction

In today's world, software has taken on a central role in nearly all parts of our daily lives, manifesting itself in an ever increasing demand for reliable and efficient systems. Yet building such systems is a complex task that traditionally requires a high level of programmer discipline in order to be robust, maintainable and extensible. Naturally, the question arises whether the underlying language can support the programmer in this task, by making well-architected solutions easy to express and concise.

Functional programming is a proven paradigm that aims to make this possible by letting users write declarative, modular and composable code through extensive use of pure functions. Besides encouraging programmers to be explicit about data, behavior and effects, functional programs are easier to reason about both for humans and machines, thus often providing opportunities for compilers to apply deep optimizations throughout a program, both in time and space.

Functional logic programming takes this approach to the next level by introducing concepts from logic programming, such as constraint solving, nondeterminism and free variables, making it particularly suitable for solving optimization problems, mathematical puzzles and more.

The *Curry* programming language implements the functional logic paradigm by extending the syntax of the purely functional language Haskell with nondeterminism, constraint solving and other logic programming techniques [Han16]. Classically, implementing a language like Curry has been a major challenge in that large parts of the compiler had to be written from scratch, even when targeting a high-level language such as Haskell. Recent work by Prott has shown, however, that extending the existing *Glasgow Haskell Compiler (GHC)* with only the Curry-specific functionality through a plugin is feasible by implementing a transformation from nondeterministic Curry code to monadic Haskell code [Pro20]. While nondeterminism constitutes one of the core features of Curry, modern Curry systems offer a wide range of other features that are currently missing from the plugin's implementation. This includes building blocks for logic-oriented programs, such as free variables or unification, which let the programmer write code in a style similar to Prolog, as well as support for *Input/Output (IO)* and encapsulated nondeterministic searches. While the latter two features can be worked around by performing IO and encapsulation outside of plugin-processed modules, a fully-featured implementation of Curry should make it possible to write the entire program within it. Lastly, *GHC's interactive shell (GHCi)* is too inconvenient for general use with the plugin, in particular due to its lack of support for the plugin's transformations in the interactive context.

1.1 Contributions

In this thesis we therefore extend the existing Curry plugin in four concrete ways to turn it into a general-purpose Curry system:

▷ First, we add free variables and unification as proposed in section 3.1.

▷ Secondly, we add support for IO functions in Curry modules as proposed in section 3.2.

- ▻ Thirdly, we provide encapsulated search through set functions as proposed in section 3.3.
- ▻ Finally, we implement a *Read-Eval-Print-Loop (REPL)* as a convenient and versatile user interface to this new Curry system as proposed in section 3.4.

1.2 Outline

We begin by introducing the necessary preliminaries in chapter 2. This includes a short introduction to the Curry programming language, the GHC *Application Programming Interface (API)* and the design of the Curry plugin. Then we discuss the user-facing design of our extensions to the Curry plugin in chapter 3 and present their implementation in chapter 4. Finally, we evaluate the implementation with regard to aspects such as completeness, extensibility, maintainability and performance in chapter 5 and conclude the thesis with a summary and suggestions for future work in chapter 6.

Chapter 2

Preliminaries

We begin by introducing the languages, concepts and APIs needed to understand the design and implementation of this new Curry system. Throughout the thesis, we will assume that the reader is already familiar with the general concepts of functional programming and basic Haskell syntax.

2.1 Curry

Curry is a functional logic programming language with a syntax closely inspired by Haskell. By seamlessly integrating the purely functional paradigm with nondeterminism, free variables and other features classically known from logic programming languages, it lets the programmer write highly declarative, concise and well-abstracted code [Han16]. While logic programming languages like Prolog require the programmer to think in terms of predicates throughout the program, Curry naturally extends the functional programming style known from Haskell, thereby providing a powerful framework for applying concepts from functional, logic and constraint programming in a wide variety of domains, including compilers [BHPR11], algorithms [Han], distributed [Han99], web [HK10] and graphical applications [HK08].

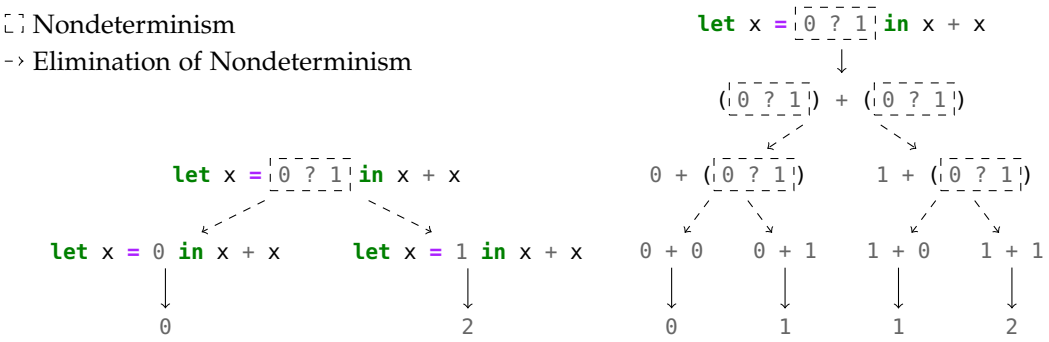
2.1.1 Nondeterminism

In the context of functional logic programming with Curry, *nondeterminism* refers to the ability of expressions to evaluate to multiple or no values. More concretely, *nondeterministic functions* are relations rather than functions in the mathematical sense, since they no longer have to unambiguously relate an input value to a single output value.

While languages like Haskell can be used to express nondeterminism, they generally require the programmer to spell it out explicitly, for example by requiring manual use of a nondeterminism monad. Curry's nondeterminism on the other hand is *ambient*, i.e. invisible to the programmer. Concretely, this means that nondeterminism uses the same syntax for function application, abstraction and variable binding as deterministic functions. One concrete way in which nondeterminism can occur in Curry programs is through the specification of non-exhaustive or overlapping patterns in function rules. Consider the following example, which models a coin that can take on either of two values, zero or one:

```
coin :: Int
coin = 0
coin = 1
```

This example is not valid Haskell, since we have two rules for the function `coin`. In Curry, however, such definitions are allowed and cause the runtime to search for all possible solutions when evaluating the expression `coin`, in this case `0` and `1`. Since the introduction of nondeterminism is a very common



(a) Call-time-choice **(b)** Run-time-choice

Figure 2.1. Schematic evaluation using call-time-choice and run-time-choice

operation in Curry programs, the Curry Prelude provides the binary operator `(?)`, also referred to as the *choice operator*, whose application nondeterministically evaluates to one of its arguments:

```
(?) :: a -> a -> a
x ? _ = x
_ ? y = y
```

We could now rephrase our previous example slightly more concisely by defining `coin = 0 ? 1`.

Nondeterminism and Laziness

Like Haskell, Curry is *lazy* and therefore only evaluates expressions as needed. This makes it possible to represent recursive values such as infinite lists, since terms are only evaluated as far as another function pattern-matches them. While laziness allows more programs to terminate than in a strict language, there are some subtle interactions with nondeterminism that we will briefly discuss in the following paragraph.

Perhaps surprisingly, nondeterministic expressions in the context of laziness cannot always be assigned an unambiguous set of values without first specifying an evaluation strategy [HA77]. This is a key point to note since it stands in contrast to Haskell's referential transparency that would allow us to replace every use of a variable with its definition. Consider the following example:

```
let x = 0 ? 1 in x + x
```

There are two major ways in which such an expression could be interpreted, as illustrated in figure 2.1: The first would be to evaluate it as `(0 ? 1) + (0 ? 1)`, yielding `0`, `1`, `1` and `2` as results. This strategy is referred to as *run-time-choice*, since we defer the nondeterminism to the point where we actually evaluate the nondeterministic value, in this case `x`. The other would be to evaluate it as `(0 + 0) ? (1 + 1)`, resulting in `0` and `2`. The latter strategy is named *call-time-choice* as it commits to the nondeterministic choice of `x` as soon as the variable is bound [HA77]. We also say that the choice is *shared* between the occurrences of `x`. While run-time-choice is easier to implement, Curry uses call-time-choice, since the semantics of nondeterministic programs then align with their strictly evaluated interpretation and many computations are more naturally expressed with it [Han16; FKS11].

2.1.2 Free Variables and Unification

Another common feature of functional logic languages is their support for *free variables*, which refer to variables that are not bound to a value. Once an expression is evaluated, the runtime tries to find a valid assignment for every free variable that occurs in it. While logic programming languages like Prolog generally treat every variable as free, Curry as a language rooted in the functional paradigm supports both bound and free variables. Unlike Prolog, Curry does not introduce free variables implicitly, however, and requires them to be declared using the `free` keyword.

Nondeterminism and free variables are closely related. Consider the following example:

```
unknown :: Data a => a
unknown = x
  where x free
```

Here, `unknown` is a function that nondeterministically returns an arbitrary value of type `a`[1], e.g. `True` and `False`, if we instantiate `a` to `Bool`. Free variables are thus another way to introduce nondeterminism into Curry programs.

To be truly useful, however, we need a way to constrain such values. For this purpose, Curry provides several unification operators, notably including `(=:=)`, the *strict equality*[2]. The application of this operator evaluates to `True` if `x` and `y` are strictly unifiable, i.e. reducible to the same term, and fails otherwise, making it especially useful for use as an equational constraint, for example in pattern guards. The following snippet illustrates this:

```
invert :: (Data a, Data b) => (a -> b) -> b -> a
invert f y | y =:= f x = x
  where x free
```

This `invert` function takes another function and emits its inverse, by using a free variable to represent a possible input `x` for a given output value `y`. Indeed, if we define

```
anyOf :: [a] -> a
anyOf = foldr1 (?)
```

as the function that nondeterministically picks an element from a list and evaluate `invert anyOf True`, the Curry REPL enumerates every possible list of type `[Bool]` that contains `True`.

2.1.3 Implementations

There are several major implementations of Curry, notably including the *Portland Aachen Kiel Curry System (PAKCS)*, the *Kiel Curry System Version 2 (KiCS2)* and *Curry2Go*, which share a common frontend. While PAKCS translates Curry source code into Prolog [AH00], KiCS2 emits Haskell code that can further be compiled into native binaries using the GHC [BHPR11] and Curry2Go uses the intermediate *ICurry* format to compile Curry programs into Go programs. A visual overview of the Curry compiler infrastructure can be found in figure 2.2.

[1] The `Data` context can be ignored for now, we will take a closer look at it later and only include it for completeness here.

[2] Curry actually has multiple strict equality operators, the other one being (===), which is in essence a deterministic variant of `(=:=)` that returns `False` instead of failing nondeterministically. The specifics are not relevant for now, we will take a closer look at this operator later.

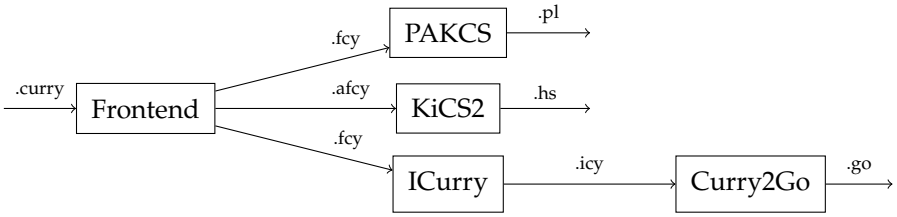


Figure 2.2. Curry compiler infrastructure

2.2 Glasgow Haskell Compiler

GHC is a compiler that translates Haskell to native code. Initially developed as a research project in academia, GHC has become one of the most widely used Haskell compilers today. Besides implementing the Haskell 2010 standard, GHC supports many language extensions that further extend Haskell's type system or provide facilities for metaprogramming.

Architecturally, GHC uses a fairly standard compilation pipeline that parses, desugars, optimizes and generates code, translating Haskell source code into a variety to intermediate formats before emitting machine code or LLVM IR, as illustrated in figure 2.3 [MPJ12].

GHC is a *bootstrapped* compiler, i.e. it itself is written in Haskell. Since GHC is a regular Cabal package, it is possible to integrate the compiler's modules into other projects as a library. Most notably this includes compiler plugins, which are dynamically loaded by GHC's driver and extend the compiler with custom functionality, e.g. new compilation passes.

2.2.1 Plugin API

To support plugins that consume GHC as a library, GHC offers a plugin API which provides extension points for injecting custom passes into the various stages of compilation [PWN19; Tea20]. For the purposes of this work, we will mostly focus on plugins hooking into the frontend-related stages, i.e. transformations that operate on abstract representations of Haskell source code. These plugins are called *source plugins*. While it is also possible to operate on GHC's lower-level intermediate languages, we will not discuss such plugins in detail.

Source plugins operate on the same internal representation of the Haskell *Abstract Syntax Tree (AST)* as the compiler itself, which is parameterized over the compilation phase to accommodate for the information added in each phase, e.g. resolved names or type annotations [NPJ17].

(1) Parser Plugins

Parser plugins operate on the AST that represents identifiers using `RdrName`, i.e. a name whose 'kind' (unqualified, qualified) is known, but not much more.

(2) Renamer Plugins

Renamer plugins operate on the AST that represents identifiers using `Name`. These names are no longer ambiguous and uniquely identify e.g. the variables or types they reference, including the module they originate from.

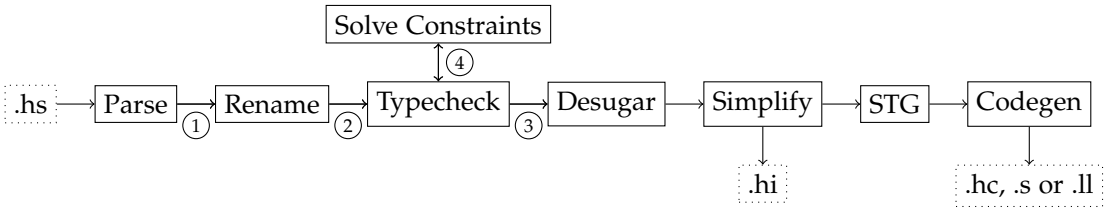


Figure 2.3. Simplified GHC compilation pipeline with extension points

③ Type Checker Plugins

Type checker plugins operate on the AST that represents identifiers using `Id` and includes type annotations. The Curry plugin that we will discuss below mainly operates in this phase, since the monadic transformation implemented by the plugin requires accurate type information.

④ Constraint Solver Plugins

Since type classes are a core feature of Haskell, a major task of the typechecker is to resolve constraints. A *constraint* formally denotes a predicate on types, commonly an instance of a type class, e.g. `Show a`, or an equality [SJSC08], e.g. `a ~ Int`. Constraint solver plugins can hook into the typechecker's internal constraint solver, by providing a function for simplifying or rejecting a set of constraints.

2.3 Curry Plugin

Although Curry can be regarded as a near-superset[3] of Haskell, its current compiler infrastructure as sketched in figure 2.2 requires implementors to write new language features and extensions largely from scratch. For this reason, a new approach has been explored that integrates Curry's nondeterminism in its various forms, including the choice operator and non-exhaustive pattern matching, into the existing GHC infrastructure through a plugin [Pro20]. By reusing most of GHC's compilation stages, the Curry plugin lets us support language extensions like `MultiParamTypeClasses` or `FunctionalDependencies` without much additional effort.

The Curry plugin hooks into the extension points ② and ③ to handle imports of other Curry modules correctly, as well as ④ for the transformation of potentially nondeterministic Curry declarations into standard Haskell declarations at the heart of the plugin, referred to as *monadic lifting*, or *lifting* for short. During the lifting, function, type, class and instance declarations are transformed into a monadic representation. We will take a closer look at the lifting algorithm in the following.

2.3.1 Lifting

To represent nondeterminism in Haskell, the plugin follows the approach presented in [FKS11] by defining the `Nondet` monad to represent nondeterministic values along with a `MonadPlus` instance for combining them and to fetch the empty computation. Informally, the `mplus` and `mempty` operations are thus analogous to Curry's `(?)` and `failed`, respectively.

The main lifting is based on the call-by-name translation scheme from [Wad90] and is performed in several phases. In the first phase, data type declarations are lifted by generating an `ND`-postfixed

[3] Technically, it is not a superset, it generally strives to support all major language features of Haskell though.

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

(a) Curry declaration

```
data ListND a = NilND
              | ConsND (Nondet a)
                       (Nondet (ListND a))
```

(b) Lifted declaration

Figure 2.4. Monadic lifting of the `List` type

```
twice = x + x
  where
  x = 0 ? 1
```

(a) Curry declaration

```
twice = (+) >>$ x >>$ x
  where
  x = (?) >>$ return 0
          >>$ return 1
```

(b) Lifted declaration without sharing

```
twice = share x >>= \x' ->
        (+) >>$ x' >>$ x'
  where
  x = (?) >>$ return 0
          >>$ return 1
```

(c) Lifted declaration with sharing

Figure 2.5. Monadic lifting of a function with and without sharing

version of every type, wrapping each constructor's arguments in `Nondet`. For example, the `List` type would be lifted as shown in figure 2.4. The plugin then derives internal instances for the lifted types, we will see why this is important later on.

Type expressions are lifted similarly: They are wrapped in `Nondet`, then recursively the type constructor and its arguments are lifted. For example, the type `[Int]` would be lifted to `Nondet (ListND Int)`. Note that `Int` as a 'primitive' type is not lifted. Arrow types are lifted similarly, e.g. `Int -> Int` would become `Nondet (Nondet Int -> Nondet Int)`. For convenience, the plugin defines the type synonym

```
type a --> b = Nondet a -> Nondet b
```

This lets us write the lifted version of `Int -> Int` as `Nondet (Int --> Int)`.

After types have been lifted, the plugin proceeds to apply the algorithm from [Han19] for transforming pattern-matching function rules into lambdas and nested case expressions as a form of preprocessing and finally lifts the function implementations themselves. The main idea is to wrap lambdas in `return`s and to bind values using `(>>=)` before applying arguments or pattern-matching on them. For better readability of lifted values, we will introduce the monadic application operator `(>>$)`:

```
infixl (>>$)
(>>$) :: Monad m => m (a -> m b) -> a -> m b
mf >>$ x = mf >>= ($ x)
```

Specialized to `Nondet` we can view this function as `(>>$) :: Nondet (a --> b) -> (a --> b)`, making it particularly useful to write monadic application chains more readably:

```
  (?) >>= \f -> f (return 0)  >>= \f' -> f' (return 1)
= (?) >>=       ($ (return 0)) >>=        ($ (return 1))
= (?) >>$           return 0   >>$            return 1
```

While this transformation takes care of representing nondeterminism in general, Curry's call-time-choice semantics require us to *share* choices as described in section 2.1.1. For this reason, the plugin adopts the approach from [FKS11] by introducing a type class `Shareable` that is adopted by values that can be shared and by updating the function lifting to insert `share` calls automatically, as depicted in figure 2.5.

Chapter 3

Design

The central design goal of this thesis is to extend the Curry plugin to a full Curry system with support for free variables, IO, encapsulated search and to provide an interface in the form of a REPL with a focus on convenience and usability. In this section we will take a closer look at the user-facing design of this new Curry system.

3.1 Free Variables and Unification

Free variables and unification as described in section 2.1.2 are essential features of Curry and an important part of making the language expressive, especially in the domain of programs that make use of logic and constraint solving techniques. While classic Curry compilers such as PAKCS or KiCS2 use the `free` keyword to introduce free variables, a limitation of GHC prevents us from extending the Haskell syntax like this with the plugin [Pro20]. Fortunately, this restriction turns out not to be a big issue, since there are ways to express free variables purely in terms of existing language concepts, as detailed in the next section.

3.1.1 Data Type Class

To implement free variables and unification, the involved types have to satisfy two fundamental properties: Enumerability and a notion of equality. Since free variables intuitively defer the responsibility of binding a value to the evaluation environment, the latter needs to be able to search for a matching value, which is only possible if there is a way to enumerate every value of the corresponding type. The main usefulness of free variables derives from the ability to constrain them, therefore we also provide a unification operator which compares values by term-level equality.

The approach we will take is to introduce a new type class `CurryData`[1] as proposed in [HT20a], which describes precisely these two operations:

```
class CurryData a where
  aValue :: a
  (===)  :: a -> a -> Bool
```

Informally, a type that conforms to `CurryData` represents anything that resembles 'data', i.e. values of a purely algebraic data type or a primitive such as `Int`. These types provide a notion of equality and are enumerable, making them suited for use as free variables, as established above. Note that functions do not fall into this category: They are not countable in general and comparing them for equality is often an undecidable problem. For this reason we do not provide `CurryData` instances for arrow types[2].

[1] We name the type class `CurryData` instead of `Data` to avoid potential ambiguities with `Data.Data` from Haskell's `base` libraries.

[2] While there are certain kinds of functions that would admit such an instance, particularly those with a finite domain, we will not focus on these cases.

```
last :: Data a => [a] -> a
last xs | (xs' ++ [x]) =:= xs = x
  where xs', x free
```

(a) Traditional syntax

```
last :: CurryData a => [a] -> a
last xs | (xs' ++ [x]) =:= xs = x
  where { xs' = aValue; x = aValue }
```

(b) `CurryData`-based syntax

Figure 3.1. Expressing free variables in terms of `CurryData`

The `CurryData` type class is automatically derived by the compiler for both built-in and user-defined algebraic data types and provides two operations: `(===)`, the strict equality, and `aValue`, which nondeterministically evaluates to an arbitrary value of the implementing type. The latter is particularly interesting since it lets us write free variables without any special syntax as shown in figure 3.1. This approach is called *free variables as generators* and `aValue` is therefore also known as a *generator function* [AH06b]. Using generator functions to model free variables, however, requires generators to indeed evaluate to every ground (i.e. variableless) term of the data type in question, in which case the generator is said to be *complete*. [AH06b] proves that the compiler-derived `aValue` implementation, which we will introduce later, is complete, therefore we can safely adopt this approach[3].

`(===)` also serves an important purpose, however: It is an equality operator that, unlike `Eq`, always matches the compiler-derived equality[4] and is thus suitable for use as an unification operator. Using `(===)` we can define the other strict equality operator `(=:=)`, which we introduced earlier in section 2.1.2:

```
(=:=) :: CurryData a => a -> a -> Bool
x =:= y | x === y = True
```

3.2 IO in Nondeterministic Contexts

Since Curry is a language designed to look and feel like a superset of Haskell, IO is an essential primitive for writing programs that interact with the user or interface with files. Until now, however, IO was not available within Curry contexts, requiring the user to write IO code outside of plugin-compiled modules and to manually deal with the bridging between the nondeterministic (lifted) and the deterministic world of Haskell. Ideally, it should be possible to write programs entirely within the realm of the plugin, both for usability and for expressiveness.

We therefore add the `IO` monad to the plugin's built-ins, making it available from within nondeterministic modules and provide lifted wrappers for a range of common operations, including the following, known from Haskell's `Prelude`:

```
putChar     :: Char   -> IO ()
putStr      :: String -> IO ()
putStrLn    :: String -> IO ()
getChar     :: IO Char
getLine     :: IO String
getContents :: IO String
```

[3] While semantically correct, a native implementation of free variables and unification within the plugin would be more performant than using generators and strict equality. See section 5.4.1 for further discussion.

[4] Technically, there is no restriction that prevents users from writing a custom instance of `CurryData`. Practically, however, this would almost always result in an 'overlapping instances' error for Curry types, since the compiler already provides instances for these.

While the lifting of pure functions has clearly defined semantics, the lifting of IO actions is a bit more involved, since they may perform side effects. Importantly, it raises the question on how to proceed with nondeterministic input values, as for example in `putStrLn ("hello" ? "world")`. While we could perform a *pull-tabbing step* [HT20b] here to move the choice towards the root of the expression, i.e. interpret it as `putStrLn "hello" ? putStrLn "world"` and evaluate it like `putStrLn "hello" >> putStrLn "world"`[5], nondeterminism, laziness and IO can interact in surprising and often unexpected ways. To illustrate this, consider another example:

```
putStr (('a' : failed) ? "b")
```

To evaluate it, we could perform a pull-tabbing step again. Since `putStr` evaluates the string lazily, `putStr ('a' : failed)` would print the `'a'` before failing, thus evaluating the entire expression would output `'ab'`. On the other hand, a strict interpretation of the expression, i.e. evaluating the argument before printing it, would yield us only `'b'` as an output, since `(('a' : failed) ? "b")` evaluates to `"b"`. This violates our assumption that Curry's nondeterminism up to differences in termination behavior can be interpreted strictly, as established in section 2.1.1.

Existing Curry compilers such as KiCS2 or PAKCS take a different approach and forbid IO entirely[6] by throwing an error whenever IO depends on a value that is nondeterministic at runtime or IO actions are combined using `(?)`. This behavior of throwing an error at runtime whenever nondeterminism and IO are intermixed is also documented in section 7.1 of the Curry report [Han16].

We therefore follow the precedent of the report and the other compilers, by requiring values used in IO actions to evaluate to a unique value and by throwing an error otherwise. Concretely, this means that evaluating expressions like `putStrLn ("hello" ? "world")` will throw an error at runtime[7]:

```
> putStrLn ("hello" ? "world")
Main: Nondeterminism in IO is not supported
```

3.3 Encapsulated Search

While IO is useful for writing user-facing programs in Curry, not being able to use it in conjunction with nondeterminism turns out to make it less useful e.g. in logic-oriented programs that traditionally make heavy use of nondeterminism in its various forms. *Encapsulated search* offers a way to perform separate searches over nondeterministic values and to collect the results in a deterministic value [BHH04]. If we consider the definition `coin = 0 ? 1` from our examples in the beginning, encapsulated search allows us to retrieve the different choices as a list, e.g. using `allValues coin`, which evaluates to `[0, 1]`. This is a powerful concept, as it allows us to reason about the results of nondeterministic computations from within another possibly nondeterministic computation, something that cannot be done with only Curry's ambient nondeterminism.

We have to clarify what `allValues` actually encapsulates, however. In the presence of Curry's call-time-choice semantics, there are several ways to interpret an expression such as the following:

```
let coin = 0 ? 1 in allValues coin ++ allValues coin
```

[5] Such a semantic would additionally imply fixing a search strategy.

[6] PAKCS is still comparatively liberal in the nondeterminism it accepts within IO, permitting examples such as `putStr (('a' : failed) ? "b")`. This is an implementation detail, however, and such expressions should not be used in actual Curry programs.

[7] We will not forbid expressions from introducing nondeterminism outside of IO actions due to the way the choice operator is implemented internally. The expression `putStrLn "hello" ? putStrLn "world"` will for example not throw an error when evaluated. Such expressions should be avoided, however, as their semantics depend on the choice of search strategy.

[BHH04] describes two fundamental forms of encapsulation, *strong* and *weak encapsulation*. Strong encapsulation encapsulates all nondeterminism that occurs in the argument. With a strongly encapsulating `allValues`, the expression would be equivalent to `allValues (0 ? 1) ++ allValues (0 ? 1)` and thus `[0, 1, 0, 1]`. Weak encapsulation, on the other hand, does not encapsulate shared choices. The expression would behave like `(allValues 0 ++ allValues 0) ? (allValues 1 ++ allValues 1)` and yield both `[0, 0]` and `[1, 1]`.

Neither of these strategies can be considered universally better. Contrary to its intuitive definition, strong encapsulation has the problem of relying heavily on details in the evaluation behavior. Another example from [BHH04] illustrates this:

```
let coin = 0 ? 1 in (allValues coin, [coin], allValues coin)
```

While the first `allValues coin` will fully encapsulate the `coin`, the use in `[coin]` forces the runtime to commit to one of the choices, yielding `([0, 1], [0], [0])` and `([0, 1], [1], [1])`. Weak encapsulation, on the other hand, does not capture variables originating from an outside context, making it hard to encapsulate locally bound variables, such as the coin in this case.

3.3.1 Set Functions

The approach presented in [AH09] therefore introduces a new form of encapsulation, so-called *set functions*. Let $\mathcal{P}(X)$ be defined as the power set of an arbitrary set X. For every function $f : a_1 \to ... \to a_n \to b$, we define an associated set function[8] of the form $f_S : a_1 \to ... \to a_n \to \mathcal{P}(b)$ that encapsulates precisely the nondeterminism that is introduced within the function, not however nondeterminism in the arguments, i.e. from the 'outside'. Thus the set function operator $\cdot_S : (a_1 \to ... \to a_n \to b) \to a_1 \to ... \to a_n \to \mathcal{P}(b)$ can be considered a hybrid approach between strong and weak encapsulation, since it strongly encapsulates the function given as a first argument and does not encapsulate the remaining arguments. These are often the desired semantics, since they accurately reflect the primary motivating use case of encapsulation: Lifting nondeterministic computations to the 'deterministic' world.

It should also be noted that a strongly encapsulating `allValues` can be viewed as a special case of the set function operator where $n = 0$.

The set operator is a *polyvariadic function* [Kis], i.e. a function that is not only polymorphic in the argument types itself, but also in the *number* of arguments. This makes it tricky to model in vanilla Curry without resorting to language extensions like multi-parameter type classes (with functional dependencies), therefore the standard `setfunctions` package for Curry provides only a finite number of fixed-length set functions:

```
set0 :: b -> [b]
set1 :: (a1 -> b) -> a1 -> [b]
set2 :: (a1 -> a2 -> b) -> a1 -> a2 -> [b]
...
set7 :: (a1 -> a2 -> a3 -> a4 -> a5 -> a6 -> a7 ->  b)
     ->  a1 -> a2 -> a3 -> a4 -> a5 -> a6 -> a7 -> [b]
```

[8] We informally identify sets with types here. While there are some subtle differences, they can be considered equivalent for our purposes. Additionally, the usual mathematical formalization of the set operator returns sets instead of lists. These approaches are not semantically equivalent: In a set-based, mathematical context `0 ? 0` would be equal to `0`, whereas most Curry implementations will yield two values nondeterministically. In practice, however, most techniques in one of these two formalizations are easily expressed in the other, therefore we generally choose the most convenient one.

Since the variadic approach turns out to be hard to model even given the availability of multi-parameter type classes in the plugin[9], we will implement a similar addition to the Curry plugin, making a fixed number of set functions available for use in Curry modules.

3.4 REPL for Nondeterministic Expressions

A key component of Curry systems is the REPL. In recent years, REPLs have become increasingly popular, being integrated even in major compiled programming languages such as Java [Fie14], and as such are an essential part of the developer experience. Especially in the realm of functional languages, they often are the face of the compiler, and therefore require careful focus. In our case, this includes making it as convenient as possible to perform a range of common operations that a programmer may want to do, including loading modules, evaluating expressions, saving compilations and even rebuilding the plugin. Following the precedent of other Curry compilers, we will align the syntax and commands closely with the REPLs from PAKCS, KiCS2 and even GHCi, to make the interface both familiar and compatible with existing workflows.

3.4.1 Motivation

While GHC already provides a REPL with GHCi, there are some caveats to using it with the Curry plugin. Notably, the plugin does not support lifting interactively declared values and declarations, therefore the user would have to move them into an actual Haskell file and load them via `:l`. Secondly, the user still has to manually use a Template Haskell-based evaluation function to perform a search over the nondeterministic computation they want to evaluate, leading to a rather verbose procedure, especially if they only want to evaluate a single expression. To evaluate an expression such as `let x = 0 ? 1 in x + x`, the user would have to declare the following module:

```
-- Expr.hs
{-# OPTIONS_GHC -fplugin Plugin.CurryPlugin #-}
module Expr where
myExpr = let x = 0 ? 1 in x + x
```

Then, they would have to load it into GHCi and manually perform a search over the lifted `myExpr`:

```
ghci> :set -XTemplateHaskell
ghci> :l Expr
ghci> import Plugin.CurryPlugin.Eval
ghci> $(evalGeneric BFS 'myExpr)
[0,2]
```

For this reason, we will implement a new REPL that lets the user evaluate such expressions directly:

```
> let x = 0 ? 1 in x + x
0
2
```

[9] More details on this can be found in section 6.2.1.

3.4.2 Commands

Besides being able to evaluate nondeterministic expressions, the REPL will offer a range of commands, many of which are borrowed from other Curry compilers, such as PAKCS or KiCS2, and originate from GHCi.

General Commands

At the most general level, the REPL will include commands for querying meta information about the Curry system, e.g. for listing the available commands or fetching the version, as well as a command for exiting it:

```
:help         - Lists available commands.
:quit         - Exits the REPL.
:version      - Fetches the compiler and plugin versions.
```

For convenience, `:?` will be aliased to `:help`. Additionally, the user will be able to run shell commands from within the REPL by prefixing them with `:!`.

Loading Modules

Many projects are modular and split across several files, therefore it is often useful to compile existing Curry modules and make them available inside the interactive REPL context. We will offer the familiar set of commands, known from PAKCS and KiCS2, for precompiling modules for interactive use:

```
:add          - Adds one or more modules to the loaded modules.
:load         - Loads one or more modules.
:modules      - Lists the loaded modules.
:reload       - Reloads the loaded Curry modules.
```

All commands may be abbreviated by the user as long as they are unambiguous, e.g. `:l` will be equivalent to `:load` and `:q` can be used in place of `:quit`. While reloading modules is a very common operation, `:r` is unfortunately ambiguous due to the presence of `:rebuild`, which we will introduce later. Therefore we will provide an explicit disambiguation of `:r` to `:reload`, thereby making it behave analogously to GHCi and the classic Curry compilers.

Compiling and Evaluating Expressions

At the heart of the REPL is the evaluation of expressions. Since we expect this to be the most common operation, simply typing a Curry expression will evaluate it in the REPL. For scripting purposes we will offer an `:eval` command that does the same, though. The available evaluation-related commands are as follows:

```
:compile      - Compiles a Curry expression.
:eval         - Evaluates a Curry expression.
:mtype        - Outputs the inferred monadic (lifted) type for a Curry expression.
:save         - Saves the evaluation of a Curry expression, main by default.
:type         - Outputs the inferred type for a Curry expression.
```

Like GHCi and the other Curry REPLs we will provide `:type`, abbreviated to `:t`, for outputting the inferred type of an expression[10]. Partially applying the choice operator to a string, for example, will result in the following type:

```
> :t (?) "abc"
(?) "abc" :: [Char] -> [Char]
```

Although the unlifted type is usually what the user is interested in, it can sometimes be useful to inspect the actual type after the monadic lifting. For this reason, we will also provide a new command `:mtype`, abbreviated to `:mt`, for viewing the monadic type of an expression:

```
> :mt (?) "abc"
(?) "abc" :: Nondet (Nondet (ListND Char) -> Nondet (ListND Char))
```

Aside from the ability to emit evaluations and types, our REPL will feature a `:save` command, like PAKCS and KiCS2, for saving the evaluation of an expression to an executable. This is especially useful in larger Curry programs with IO, since the resulting binary can be executed like a normal program on any[11] machine without requiring an installation of a Curry system.

Configuring the REPL Environment

Besides the core functionality of compiling and evaluating expressions, the REPL will also feature extensive customization options, prominently including the ability to set custom GHC flags and to choose the search strategy:

```
:clear        - Clears bindings, GHC flags and loaded modules.
:flags        - Lists or overwrites additional GHC flags for all invocations.
:set          - Adds GHC flags for the Curry module.
:strategy     - Sets the search strategy.
```

The search strategy determines how the tree of nondeterministic choices from an expression is traversed during evaluation. In our REPL we will support *Breadth-First Search (BFS)* and *Depth-First Search (DFS)*, like the plugin currently does.

Other Commands

Although not specific to general development with Curry, a REPL is also very useful for developing and testing the plugin itself. During plugin development, we find ourselves frequently rebuilding the Curry plugin, resulting in many 'context switches' between the REPL and the shell. To make this a more seamless experience, we will offer a command that calls `stack build` in the plugin directory for us, so we no longer have to leave the REPL:

```
:rebuild      - Rebuilds the Curry plugin.
```

[10] Note that our implementation will use Template Haskell's type reflection and therefore will not support polymorphic types and flags such as `-fprint-explicit-foralls` for now. We will go into more detail later in section 4.4.3.

[11] The binary is usual machine code, therefore it will run on any machine with a compatible CPU and OS.

Chapter 4

Implementation

In this chapter we will discuss the techniques used to implement the previous chapter's design goals. Most of the features mentioned will be implemented in Haskell directly as part of the plugin, keeping it cohesive and integrable as a standalone component in larger Haskell projects. This approach stands in contrast to implementing new features as standalone desugarings, which would have introduced more moving parts into the pipeline along with the inherent complexity of performing source-to-source transformations outside of a compiler context. The REPL, however, will be written as an external program that calls GHC and the plugin as needed.

4.1 Data Deriving

As specified in section 3.1.1, our goal is to provide the `CurryData` type class for both user-defined and built-in types. To do so, we will first introduce a scheme for deriving `CurryData` for arbitrary algebraic data types and then implement it using GHC's metaprogramming facilities.

4.1.1 Scheme

The general strategy for deriving a `CurryData` instance has been outlined in [AH06b] and [HT20a] and can be described as follows: For any algebraic data type

$$
\begin{aligned}
&\texttt{data } D = C_1 \ a_{1,1} \ \dots \ a_{1,n_1} \\
&\qquad\ \mid \dots \\
&\qquad\ \mid C_m \ a_{m,1} \ \dots \ a_{m,n_m}
\end{aligned}
$$

we can define a valid instance of `CurryData` using the following scheme:

$$
\begin{aligned}
&\texttt{instance CurryData } D \texttt{ where} \\
&\quad \texttt{aValue} = C_1 \ \texttt{aValue} \ \dots \ \texttt{aValue} \\
&\qquad\quad \texttt{?} \ \dots \\
&\qquad\quad \texttt{?} \ C_m \ \texttt{aValue} \ \dots \ \texttt{aValue}
\end{aligned}
$$

$$
\begin{aligned}
C_i \ x_{i,1} \ \dots \ x_{i,n_i} \ \texttt{===} \ C_i \ y_{i,1} \ \dots \ y_{i,n_i} &= x_{i,1} \ \texttt{===} \ y_{i,1} \\
&\ \texttt{\&\&} \ \dots \\
&\ \texttt{\&\&} \ x_{i,n_i} \ \texttt{===} \ y_{i,n_i} \qquad \forall i \in \{1, \dots, m\} \\
C_i \ x_{i,1} \ \dots \ x_{i,n_i} \ \texttt{===} \ C_j \ y_{j,1} \ \dots \ y_{j,n_j} &= \texttt{False} \qquad \forall i, j \in \{1, \dots, m\} : i \neq j
\end{aligned}
$$

In the implementation of `aValue`, we thus nondeterministically choose one of the data type's constructors and recursively invoke the `aValue` implementation for every argument of the constructor. `(===)` is also defined recursively, matching the `(==)` implementation from the compiler-derived `Eq` instance.

4.1.2 GHC Generics

Since we want to implement a generic strategy for deriving the instance from an arbitrary algebraic data type, our first idea might be to take advantage of Haskell's standard deriving mechanism, which is used to automatically synthesize instances for classes such as `Eq` or `Show`. This deriving is, however, implemented as a transformation directly in the compiler and thus hard to extend. Fortunately, GHC offers a mechanism for *datatype-generic programming*[1] that lets us abstract over the definition of data types at the language level [MDJL10]. For example, consider the following data type:

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
```

With datatype-generic programming, our program can reason about the structure of the constructors themselves as well as meta-information, such as their names. In this case, the generic representation would describe the type as the sum of a single-argument constructor named `"Leaf"` wrapping an `a` and a constructor named `"Branch"` wrapping the product of two `Tree a` values.

Similar to other metaprogramming extensions such as `TemplateHaskell`, GHC implements generics at compile-time[2] and provides a new type class named `Generic`[3] for this purpose:

```
class Generic a where
  type Rep a :: * -> *

  from :: a -> Rep a x
  to   :: Rep a x -> a
```

Every type with an instance of the `Generic` type class can be represented generically. Since algebraic data types in Haskell are always sums and products of other types, they lend themselves to a tree-style representation where each node encodes whether its children are aggregated through choice, i.e. a sum, or composition, i.e. a product. For this reason, GHC's `Generic` class includes `Rep a`, a type-level generic representation of the algebraic data type's constructors in terms of sums, products and recursively wrapped types. Here, `Rep` is an *associated type family*. In Haskell, *type families* are in essence functions at the type-level, which are termed *associated* if they are declared as part of a type class. In our case, the `Rep` family maps types to their generic representation. For a generically representable type, the methods `from` and `to` provide the corresponding value-level isomorphism between values of the type itself and values of its generic representation.

With the `DeriveGeneric` language extension, we can let the compiler derive the `Generic` instance for a data type automatically. In our example from earlier, the following exact instance would be derived:

```
instance Generic (Tree a) where
  type Rep (Tree a)
    = D1 ('MetaData "Tree" "Module" "package" 'False)
        (C1 ('MetaCons "Leaf" 'PrefixI 'False)
          (S1 ('MetaSel 'Nothing
```

[1] Datatype-generic programming is not to be confused with 'generics' known from other programming languages, such as Java. The latter is known as *parametric polymorphism* in Haskell and refers to the ability of functions and types to take type arguments.

[2] Many other compiled languages, including Java, Go or Swift, provide similar functionality through runtime reflection. Haskell programs, however, have no deep type information available at runtime, thus a dynamic approach would not work here. The type class-based generics by GHC operate entirely at compile-time instead, which also aligns with Haskell's philosophy of verifying as much of a program as possible statically.

[3] `Generic` instances are only provided for generically representable types of kind `*`. There is another type class named `Generic1` that can be implemented or derived for types of kind `* -> *`, but we will not go into detail on this class.

```
                'NoSourceUnpackedness
                'NoSourceStrictness
                'DecidedLazy)
      (Rec0 a))
  :+:
  C1 ('MetaCons "Branch" 'PrefixI 'False)
   (S1 ('MetaSel 'Nothing
                'NoSourceUnpackedness
                'NoSourceStrictness
                'DecidedLazy)
      (Rec0 (Tree a))
   :*:
    S1 ('MetaSel 'Nothing
                'NoSourceUnpackedness
                'NoSourceStrictness
                'DecidedLazy)
      (Rec0 (Tree a))))
```

Taking apart this example, the `D1` type constructor encodes meta-information about the type, including its name, the module and package it originated from and whether it is a `newtype`. Similarly, `C1` encodes meta-information about each constructor, including its name, fixity and whether the constructor declares a record. We will ignore `S1`, another metadata type constructor, for now[4]. If we rewrite the example without the meta-information, we get a better picture of the type's compositional structure:

```
instance Generic (Tree a) where
  type Rep (Tree a)
    = Rec0 a
      :+:
      (Rec0 (Tree a) :*: Rec0 (Tree a))
```

While `(:+:)` encodes the sum of constructors, i.e. a choice, and `(:*:)` encodes the product of constructors, i.e. a composition, `Rec0`, as a variant of `K1`, encodes a wrapper around another type. The latter can thus be seen as one of the basic 'building blocks' of an algebraic data type, other ones including `U1`, the unit type, e.g. as in `data Unit = Unit`, and `V1`, the uninhabited type, e.g. `data Void`.

4.1.3 Generic Instances

To provide a `CurryData` instance for every algebraic data type we now proceed in three steps:

1. First, we provide `CurryData`-like instances for the previously introduced generic primitives.

2. Then we provide a default implementation of `CurryData` for every type with a `Generic` instance.

3. Finally we let the compiler derive `Generic` and an empty instance of `CurryData`, which in that case includes our default methods, for the lifted data type.

[4]For the curious, `S1` encodes information about record field selectors. Since this type does not declare any selectors, the meta-selectors are annotated with `'Nothing`. The remaining annotations provide information about strictness and the type's memory-level representation.

While the implementation of the instances for the generic primitives is mostly a matter of translating the derivation scheme introduced in section 4.1.1 into the appropriate generic idioms, there are a few details that we need to take into account first: Since we want to define these instances as built-ins for use in Curry modules, we need to operate on a lifted version of the `CurryData` type class:

```
class CurryDataND a where
  aValueND :: Nondet a
  (===#) :: Nondet (a --> a --> Bool)
```

The standard lifting given above is not sufficient, however, since the generic representation is parameterized itself. For this reason, we define another variant of the class that is suitable for implementation by the generic primitives mentioned above:

```
class CurryDataGen a where
  aValue' :: Nondet (a p)
  (=-=) :: Nondet (a p --> a p --> Bool)
```

Uninhabited Types

Our first instance is for `V1`, the generic representation of an uninhabited type. An uninhabited type, e.g. as defined with `data Void`, has no values[5] and thus cannot be constructed. Although such types are of limited use outside of type-level computations, they are valid algebraic data types and should be treated as such. We provide the following instance:

```
instance CurryDataGen V1 where
  aValue' = mzero
  (=-=) = undefined
```

Since our instances are defined in a normal Haskell module, we use `MonadPlus` to introduce and combine nondeterminism. The implementation of `aValue'` never yields a value and thus represents the failed computation. The implementation of `(=-=)` does not matter, since there is no way to invoke the function. For simplicity, we therefore simply throw an error in the unreachable function body.

Unit Types

`U1`, the generic representation of a unit type, is another important primitive. Unit types have a single data constructor, e.g. as in `data Unit = Unit`, and thereby admit a single value. We provide the following instance:

```
instance CurryDataGen U1 where
  aValue' = return U1
  (=-=) = liftNondet2 $ \U1 U1 -> True
```

We return this single value, often also referred to as the *unit value*, in `aValue'`. Its equality function is trivially a constant `True` as there are no other values to compare besides the unit value.

[5] Technically, every type in Haskell has bottom (`undefined`) as a value. Since bottom is semantically usually not considered 'part' of the type and throws an error upon evaluation, we do not enumerate it as part of `aValue`.

Wrapper Types

The metadata primitive `M1`, which is aliased by `D1`, `C1` and `S1`, wraps another value and thus does not require much further logic. Our instance simply delegates to the inner value which also conforms to `CurryDataGen` and performs a little bit of (un-)wrapping:

```
instance CurryDataGen a => CurryDataGen (M1 i c a) where
  aValue' = M1 <$> aValue'
  (=-=) = return $ \x -> return $ \y -> do
    M1 x' <- x
    M1 y' <- y
    apply2 (=-=) (return x') (return y')
```

`K1` also wraps another type and is usually viewed in its specialized form `Rec0`, which is the generic representation of a constructor argument. Again, we delegate both the implementation of `aValue'` and the equality:

```
instance CurryDataND a => CurryDataGen (K1 i a) where
  aValue' = K1 <$> aValueND
  (=-=) = return $ \x -> return $ \y -> do
    K1 x' <- x
    K1 y' <- y
    apply2 (===#) (return x') (return y')
```

While the implementation looks similar to the `M1` instance at first, it includes a subtle, but important difference: Here, we recurse on `aValueND` and `(===#)` from `CurryDataND`, rather than `aValue'` and `(=-=)` from `CurryDataGen`. The reason for this is that `K1` wraps the actual value rather than its generic representation. Since `CurryDataND` may itself be implemented in terms of `CurryDataGen`, this implementation may cause mutually recursive invocations, which fortunately are well supported in Haskell, however.

Sum Types

The instances for the combinators encoding composite types, `(:+:)` and `(:*:)`, are a bit more involved. For the sum combinator `(:+:)` we implement `aValue'` by nondeterministically combining a value of the left type with the right type using `Nondet`'s `MonadPlus` instance and implement the equality by matching constructors on both sides and calling recursively into the left-hand and right-hand side's `CurryDataGen` instance:

```
instance (CurryDataGen a, CurryDataGen b) => CurryDataGen (a :+: b) where
  aValue' = (L1 <$> aValue') `mplus` (R1 <$> aValue')
  (=-=) = return $ \x -> return $ \y -> do
    x' <- x
    y' <- y
    case (x', y') of
      (L1 x'', L1 y'') -> apply2 (=-=) (return x'') (return y'')
      (R1 x'', R1 y'') -> apply2 (=-=) (return x'') (return y'')
      _                -> return False
```

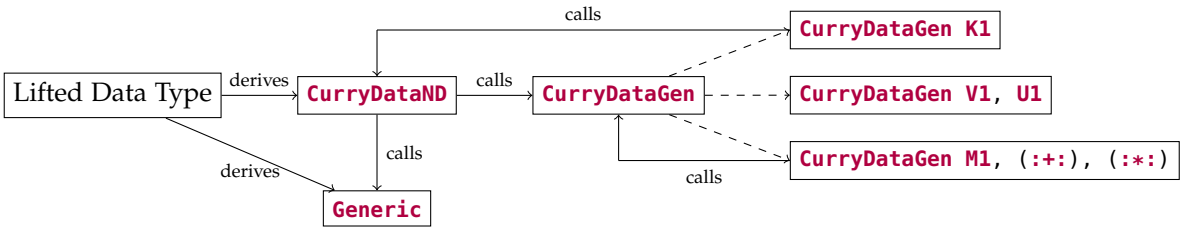


Figure 4.1. Dependency graph of generic `CurryData` implementation

Product Types

For the product combinator `(:*:)` on the other hand, we bind a value of both the left and the right type, wrap them with the lifted `(:*:)` constructor and implement the equality by comparing the left-hand sides and the right-hand sides separately:

```
instance (CurryDataGen a, CurryDataGen b) => CurryDataGen (a :*: b) where
  aValue' = (:*:) <$> aValue' <*> aValue'
  (=-=) = return $ \l -> return $ \r -> do
    x  :*: y  <- l
    x' :*: y' <- r
    xsEq <- apply2 (=-=) (return x) (return x')
    ysEq <- apply2 (=-=) (return y) (return y')
    return $ xsEq && ysEq
```

These instances cover all of our generic primitives and thus fully implement the derivation scheme introduced in section 4.1.1 for the generic representation synthesized by the compiler as part of a `Generic` instance.

4.1.4 Anyclass Deriving

To get a fully functioning `CurryDataND` instance for an arbitrary lifted data type as visualized in figure 4.1 we still need a little bit of wiring, however. First, this includes providing default implementations for `aValueND` and `(===#)` in `CurryDataND` for any type that also has a `Generic` instance. These implementations convert between the actual values and their generic representation with `to` and `from` and delegate to our `CurryDataGen` instances as shown in figure 4.2. Secondly, we need to conform the lifted type to `CurryDataND` by providing an empty instance. On the language level, we could either do this manually or use the `DeriveAnyClass` language extension. Since `DeriveAnyClass` permits arbitrary type classes in `deriving` clauses by synthesizing an empty instance for each non-standard type class, we will use the term *anyclass deriving* to refer to the practice of generating an empty instance that relies entirely on default implementations of the class's methods.

In our case, however, we want to perform this anyclass deriving implicitly, without requiring the user to add `CurryDataND` to the `deriving` clause of every new data type. The plugin already does this for a range of other built-in type classes, including `Generic`, by implementing a *derive transformation* as mentioned in section 2.3.1. This transformation generates standalone `deriving` declarations for these type classes, internally leveraging the `DeriveGeneric` and `DeriveAnyClass` extensions. Since this is exactly what we need, we add a new phase to the derive transformation that generates a `deriving` declaration for `CurryDataND` too.

```
class CurryDataND a where
  default aValueND :: (Generic a, CurryDataGen (Rep a)) => Nondet a
  aValueND = to <$> aValue'

  default (===#) :: (Generic a, CurryDataGen (Rep a)) => Nondet (a --> a --> Bool)
  (===#) = return $ \x -> return $ \y ->
    apply2 (=-=) (from <$> x) (from <$> y)
```

Figure 4.2. Default implementation of `CurryDataND` for types with a `Generic` instance

```
data ListND a = NilND
              | ConsND (Nondet a)
                       (Nondet (ListND a))

deriving instance
  ( CurryDataND a
  , CurryDataND (ListND a)
  ) => CurryDataND (ListND a)
```

(a) Lifted type with conditional `CurryDataND` instance

```
data AppND = AppND (Nondet (Int --> Int))
                   (Nondet Int)

deriving instance
  ( CurryDataND (Int --> Int)
  , CurryDataND Int
  ) => CurryDataND AppND
```

(b) Lifted type without `CurryDataND` instance[a]

[a] Note how `CurryDataND (Int --> Int)` is never satisfied

Figure 4.3. Generated deriving declarations for `CurryDataND`

Similarly to other internally derived classes such as `Shareable`, we want to make sure that `CurryDataND` is only derived for types that admit a valid instance. In our case, we want to only derive an instance if the constructors' arguments consist solely of `CurryDataND`-conforming types. To achieve this, we place constraints on the `deriving` declaration that require every argument type to have a `CurryDataND` instance, as shown in figure 4.3.

With this automatic deriving in place, lifted data types will automatically receive a valid `CurryDataND` instance, provided they satisfy the appropriate constraints.

4.1.5 Other Notes

To simplify debugging, we furthermore add a new flag named `dump-deriving-decls` that outputs the deriving declarations generated by the plugin after the derive transformation. This flag can be enabled by adding `-fplugin-opt Plugin.CurryPlugin:dump-deriving-decls` to a GHC invocation or to the REPL context by using the `:set` command.

4.2 IO Lifting

Like the implementation of the data type class and its derivation, the implementation of IO as specified in section 3.2 will, for the most part, be on the language side, i.e. in a module that is included as a `ForeignExport`. As with the other built-ins, we define them outside of a Curry module, hence our first step is to define a lifted variant of the `IO` type constructor, i.e. `IOND`, manually. `IO` is an opaque type that can be viewed as a compiler primitive, therefore we cannot lift it like a normal algebraic data type.

Since we already decided on disallowing nondeterminism in IO, a separate lifted representation is not needed and a simple `newtype` around `Prelude.IO` suffices:

```
newtype IOND a = IOND { unIO :: Prelude.IO a }
```

4.2.1 Encapsulation

We cannot feed nondeterministic values directly into IO operations and, as a consequence of our decision to require unique results, we need a way to encapsulate them. The module containing the `Nondet` monad conveniently offers an operation that does this:

```
allValues :: Nondet a -> Tree a
```

This, however, only yields us a tree, whereas we are interested in a list of values. We therefore have to decide on a search strategy that will be used to encapsulate values passed into IO functions. While both DFS and BFS are valid choices here, we decide to default to the latter as it is the more complete strategy, considering that trees of nondeterministic choices may be infinite in depth, but always branch finitely[6]. With our strategy, we can now obtain a list of values `[a]` given a `Nondet a`. Finally, we have to make sure that the result is unambiguous to be able to operate on a value of type `a` directly. For convenience, we define a function which returns the value if it is unique and outputs a readable error message otherwise:

```
requireUniqueForIO :: [a] -> a
requireUniqueForIO [x] = x
requireUniqueForIO _   = error "Nondeterminism in IO is not supported!"
```

Putting everything together, we can define a function that allows us to 'unwrap' an arbitrary nondeterministic value, which is effectively an inverse operation to the `return` method from `Monad` for unique values:

```
evalUniquelyForIO :: Nondet a -> a
evalUniquelyForIO = requireUniqueForIO . modeOp BFS . allValues
```

4.2.2 Lifting and Unlifting

While the `evalUniquelyForIO` function defined in the previous section takes care of encapsulating shallow nondeterminism, many lifted values include deeply nested nondeterminism and thus have a different representation in the deterministic world. A list of integers, for example, would be represented as `Nondet (ListND Int)` in the lifted world and as `[Int]` in the deterministic world. Considering that `ListND` may include introduce nondeterminism at the level of every constructor, we thus need to not only encapsulate the topmost layer of nondeterminism, but have to deeply evaluate all nondeterminism if we want to pass such a value to an IO function, i.e. we have to compute the value's unlifted representation.

For this, the plugin offers a *multi-parameter type class* named `Normalform` for converting between these two representations. Multi-parameter type classes are a powerful language extension to Haskell that allow us to model relations on the type-level cleanly. *Functional dependencies* additionally let us establish functional relationships between the type class's parameters, making it possible to model functions at the type-level too. In our case, this is exactly what we want: An isomorphism between

[6] Choice trees are in fact binary trees, because the choice operator `(?) :: a -> a -> a` is binary.

```
instance Normalform Nondet Int Int
instance Normalform Nondet Char Char
instance (Normalform Nondet a1 a1', Normalform Nondet a2 a2')
      => Normalform Nondet (a1 --> a2) (a1' -> a2')
instance (Normalform Nondet a1 a1')
      => Normalform Nondet (ListND a1) [a1']
```

Figure 4.4. Notable built-in instances of `Normalform`

lifted and unlifted types, along with the corresponding isomorphism over values. Since this conversion mechanism is very general, the class `Normalform` is additionally parameterized over the monad itself, leading to the following definition:

```
class Monad m => Normalform m a b | m a -> b, m b -> a where
  nf    :: m a -> m b
  liftE :: m b -> m a
```

In this class, `nf` is responsible for computing the unlifted representation of a value. Since the unlifted representation involves no nested nondeterminism and has to be of finite size, it effectively represents a *normal form* of the value, hence also the method's name. `liftE` is the inverse operation, i.e. it computes a value's lifted representation. Both operations form the isomorphism between lifted and unlifted monadic values. Analogously, the functional dependencies in the class head form the isomorphism between lifted and unlifted types.

The plugin additionally adds an associated type family for mapping lifted types to unlifted types, in essentially the same manner as the type class given above:

```
type family Unlifted m a = b
```

Using this type family we can declare some type synonyms for convenience:

```
type UnliftedN   a = Unlifted Nondet a
type NormalformN a = Normalform Nondet a (UnliftedN a)
```

These synonyms will be useful to write function signatures that involve `Normalform` contexts more compactly and with less type variables.

Internally, the plugin defines a range of built-in `Normalform` instances, notably including primitives, such as `Int` or `Char`, which have the same representation in lifted and deterministic contexts, as well as types such as functions and lists. A few examples of such instances are listed in figure 4.4. Declared data types automatically receive a derived instance as part of the same transformation that also derives other classes, such as `Shareable` or `CurryDataND`, as detailed in section 4.1.4.

Since we want to evaluate nondeterministic values uniquely to their unlifted form for use in IO, we can define a wrapper of `evalUniquelyForIO` that invokes `nf`:

```
evalUniquelyForIONF :: NormalformN a => Nondet a -> UnliftedN a
evalUniquelyForIONF = evalUniquelyForIO . nf
```

With these evaluation functions in place we can now define the actual lifting and unlifting for IO functions. Our lifting needs to differentiate between values and functions, therefore we declare these operations in a class:

```
class NormalformN a => LiftForIO a where
  liftForIO :: UnliftedN a -> Nondet a
  unliftForIO :: Nondet a -> UnliftedN a
```

For most values, the IO lifting is a simple matter of wrapping the `Normalform` instance:

```
instance NormalformN a => LiftForIO a where
  liftForIO = liftE . P.return
  unliftForIO = evalUniquelyForIONF
```

There is a catch with this instance, however, which we will run into if we instantiate `a` to a function type. Consider the following example, declared in a normal Haskell module:

```
f :: (Int -> Int) -> IO ()
f g = putStrLn (show (g 0))
```

While the lifted variant of this function, defined as `f' = liftForIO f`, will typecheck, calling `f' id` from a Curry module will result in a runtime error. In our case, the problem is that we try to unlift an application of `(->)` using its `Normalform` instance, whose implementation of `nf` is actually a partial function that throws an error once we evaluate it:

```
instance (Normalform Nondet a1 a1', Normalform Nondet a2 a2')
      => Normalform Nondet (a1 --> a2) (a1' -> a2') where
  nf mf = mf >> return (error "Plugin Error: Cannot capture function types")
  ...
```

This implementation is motivated by the problem that we cannot turn any nondeterministic function into a deterministic function in general: Values of type a_2' would need to depend directly and unambiguously on values of type a_1', which is not possible if the function relates a single input to multiple values. While our strategy of encapsulating and requiring values to be unambiguous, lets us implement such a conversion for use in IO liftings, up to throwing an error once actual nondeterminism occurs, the `Normalform` instance cannot make this trade-off. For this reason, we add another instance of `LiftForIO` that is specialized to functions:

```
instance (LiftForIO a, LiftForIO b) => LiftForIO (a --> b) where
  liftForIO f = P.return $ liftForIO . f . unliftForIO
  unliftForIO f x = unliftForIO $ f >>$ liftForIO x
```

Since this instance overlaps with the previous `NormalformN`-based instance, we have to add an `{-# OVERLAPPABLE #-}` pragma to the previous instance. Still, these overlapping instances cannot be used to lift polymorphic functions such as `fmap` as the choice of `LiftForIO` instance may depend on type variables which are first instantiated at the call site of the lifted function. For example, the arity of the mapper function in an invocation to `fmap` may depend on the type arguments to the call. This is a problem as GHC is unable to conjure the corresponding `LiftForIO` instance from the `fmap` body, given its lack of a `LiftForIO` context. To get around this limitation, we tell the compiler to default to the `NormalformN`-based instance in such cases, by annotating the instance with the `{-# INCOHERENT #-}` pragma instead. While this makes lifted polymorphic IO functions slightly more restrictive in that parameterizations over function types may result in the same `"Cannot capture function types"` error as given above, we expect this implementation to accommodate for the vast majority of use cases involving monadic IO.

4.3 Set Function Synthesis

As proposed in section 3.3.1, we want to implement set functions as a means of providing encapsulated nondeterminism from within Curry modules. Similar to the PAKCS implementation in the `setfunctions` Curry package, we will provide a fixed number of `setN` functions that return a multiset of values [Han18].

4.3.1 Scheme

As with the implementation of the `CurryDataND` class and the lifted IO functions, our implementation will take place in a built-in module that will be imported into the plugin's `Prelude` but not lifted itself.

Since our set functions should only encapsulate the first argument, i.e. the function, we will bind the remaining arguments. For the encapsulation itself, we will use `allValues` to run the `Nondet` monad and use `modeOp BFS` to perform a BFS over the resulting tree of nondeterministic choices, similarly to the implementation for IO in section 4.2.1. The choice of BFS as a default, while arbitrary, is primarily motivated by the desire to have a search strategy that is as complete as possible. Finally, the list of results from the encapsulated search is a regular Haskell list, therefore we need to lift it to the nondeterministic world. Since the `liftE` function from `Normalform` only operates on monadic values, we will provide a deterministic function `liftList :: [a] -> ListND a` for this.

With these ideas in place, our first attempt at formalizing a scheme for implementing an n-ary set function could look like this[7]:

```
setN :: ( ShareableN a1
        , ...
        , ShareableN an
        , ShareableN b
        ) => Nondet ((a1 --> ... --> an -->        b)
                  --> a1 --> ... --> an --> ListND b)
setN = return $ \f -> return $ \x1 ->
                    ...
                    return $ \xn -> x1 >>= x1' ->
                                    ...
                                    xn >>= xn' ->
  return $ liftList $ modeOp BFS $ allValues $ f >>$ return x1'
                                                 >>$ ...
                                                 >>$ return xn'
```

Here, we declare a lifted function as established in section 2.3.1 by wrapping the lambda abstractions into `return`s. Subsequently, we bind every argument except for the function to be encapsulated. This lets us reason about the remaining arguments as fixed values and causes nondeterminism in the arguments to become unaffected by our capsule.

For simple examples such as `set2 (?) 0 1`, which evaluates to `[0, 1]`, this implementation works as expected. The Curry plugin, however, lets us declare data types with deeply nested nondeterminism such as `ListND`, allowing nondeterminism to occur at the level of every constructor separately. Since we only invoke `allValues` on the top-level `Nondet` wrapper, our set function implementation leaks deeply

[7] Note that there are a few implementation details in this scheme that we will mostly skip over, since they are not needed to understand the strategy itself. The `ShareableN` constraints, for example, are required internally to make lifted polymorphic functions work correctly.

nested `Nondet` values, an issue that manifests itself in obscure and hard-to-debug runtime failures[8]. We therefore use a trick to flatten deeply nested nondeterminism into a single `Nondet` by converting both the arguments and the encapsulated function's output into the normal form and back using `liftE . nf`[9]. A nice side effect of this trick is that we can replace our `liftList` with a standard `liftE` invocation:

```
setN = return $ \f -> return $ \x1 ->
                      ...
                      return $ \xn -> liftE (nf x1) >>= x1' ->
                                      ...
                                      liftE (nf xn) >>= xn' ->
  liftE $ return $ modeOp BFS $ allValues $ nf $ f >>$ return x1'
                                                   >>$ ...
                                                   >>$ return xn'
```

While this scheme already works well for many cases, including `set2 (?) 'a' ('b' ? 'c')`, which yields `['a', 'b']` and `['a', 'c']`, getting the correct semantics when sharing a choice between the encapsulated function and the not-to-be-encapsulated arguments turns out to be slightly more challenging. Consider the following example:

```
let coin = 0 ? 1 in set1 (+ coin) coin
```

Curry's call-time choice suggests that this expression should evaluate to `[0]` and `[2]`, since we decided on the coin's value before encapsulating the function. Indeed, if we evaluate this example with PAKCS, we get the expected result. Our plugin, however, yields `[0, 1]` and `[1, 2]`.

The issue, in our case, is that the main computation and the encapsulated computation, the latter being run by `allValues`, do not share the same *store*. Stores are the data structure that the plugin uses to keep track of shared choices internally[10]. Importantly, values inside the `Nondet` monad can access an associated store through a corresponding `MonadState` instance, therefore we can fix our scheme by passing the main computation's store into the encapsulated computation with `get` and `put`:

```
setN = return $ \f -> return $ \x1 ->
                      ...
                      return $ \xn -> liftE (nf x1) >>= x1' ->
                                      ...
                                      liftE (nf xn) >>= xn' -> Nondet get >>= \store ->
  liftE $ return $ modeOp BFS $ allValues $ nf $ Nondet (put store) >>
                                                   (f >>$ return x1'
                                                      >>$ ...
                                                      >>$ return xn')
```

With the modified scheme, choices from the main computation are now available in the encapsulated computation, making our previous example evaluate to `[0]` and `[2]`, as expected.

[8] The issue is that the heaps („stores") that represent their shared choices may no longer be available in the evaluation context.

[9] This makes the set function strict in its arguments, similar to e.g. the PAKCS implementation of set functions. We will discuss this limitation in more detail later in section 5.4.2 and section 6.2.2.

[10] More specifically, a store is an untyped, labelled heap of values, the specifics are not relevant to us, however.

4.3.2 Template Haskell

Writing the set functions manually is a daunting and error-prone task, therefore we will use *Template Haskell* to write a meta function that generates our `setN` functions according to the previously introduced scheme. Template Haskell is a language extension that permits a form of metaprogramming in Haskell, similar to macros in other languages. In contrast to the preprocessor employed by C and C++, however, Template Haskell programs are not mere text substitutions, but are fully-featured Haskell functions that operate on AST nodes. This makes them a powerful abstraction for our purpose of generating function declarations.

The Template Haskell AST is described using several algebraic data types, notably including **`Dec`** for declarations, **`Pat`** for patterns, **`Exp`** for expressions and **`Type`** for types. For our purposes only **`Exp`** and **`Pat`** will be relevant, which are defined as follows:

```
data Exp = VarE Name      -- x
         | LitE Lit       -- 5
         | AppE Exp Exp   -- f x
         | LamE [Pat] Exp -- \p1 p2 -> x
         | ...

data Pat = VarP Name      -- x
         | LitP Lit       -- 5
         | ...
```

Here, **`Name`** is used to describe variable identifiers and **`Lit`** represents a literal value, such as a string, character or integer.

Since Template Haskell functions need access to compile-time meta information about the program, fresh variables and more, they use a special monad named **`Q`**, which provides this ambient state. Our `setN` meta function will therefore have the type `setN :: Int -> Q Exp`, indicating that we take an arity and generate an expression, in our case the function body. We begin by binding some variable identifiers for later use:

```
f <- newName "f"
xs <- replicateM n $ newName "x"
xs' <- replicateM n $ newName "x'"
store <- newName "store"
```

Constructing a piece of Haskell AST manually, e.g. using **`Exp`**'s constructors in a long chain of **`AppE`** and **`LamE`**s, can become very verbose. For this reason, Template Haskell provides a convenient syntax for capturing the Haskell AST of a literal expression with `[| ... |]`. We will use this to wrap the monad operators, which are central to our scheme:

```
sqc <- [| (>>) |]
bnd <- [| (>>=) |]
apm <- [| (>>$) |]
ret <- [| return |]
```

Before implementing the scheme itself, we provide a few helper functions, notably including one that produces a lifted lambda abstraction of the form `return $ p1 -> ... -> return $ \pm -> b`. This structure forms the basis of a lifted function body and can be generated as follows:

```
genNDLam :: [Pat] -> Exp -> Exp
genNDLam ps b = foldr (\p b' -> AppE ret (LamE [p] b')) b ps
```

Additionally, we declare a function that generates bindings for several monadic values in sequence, in the form of `m1 >>= \a1 -> ... mm >>= \am -> b`. For our scheme, these will be useful to bind the not-to-be-encapsulated arguments. The function can be implemented as follows:

```
genBinds :: [Exp] -> [Pat] -> Exp -> Exp
genBinds ms as b = foldr (\(m, a) b' -> foldl AppE bnd [m, LamE [a] b']) b
```

We also add a generator for lifted applications of the form `e >>$ return a1 >>$ ... >>$`, in this case motivated by the need to apply the to-be-encapsulated function to the already bound arguments:

```
genNDApply :: Exp -> [Exp] -> Exp
genNDApply e as = foldl (\l r -> foldl AppE apm [l, r]) e (AppE ret <$> as)
```

Aside from these utilities, we need generators for expressions that inject the store for nondeterministic choices into the encapsulated computation. For better readability, we first bind two helper expressions:

```
storeGetter <- [| Nondet get |]
storePutter <- [| Nondet . put |]
```

We then provide a generator for expressions that bind the store using `Nondet get >>= \store -> e`:

```
genBindStore :: Exp -> Exp
genBindStore e = foldl AppE bnd [storeGetter, LamE [VarP store] e]
```

Similarly, we provide a generator that inserts the store with `Nondet (put store) >> e`:

```
genPutStore :: Exp -> Exp
genPutStore e = foldl AppE sqc [AppE storePutter (VarE store), e]
```

Finally, before putting all of these components together, we need to generate the snippet that performs the actual encapsulation using `nf` and `allValues`, traverses the tree and lifts the results again, as well as the function that flattens nondeterminism by converting to the normal form and back. We achieve this by wrapping the following expressions:

```
finder <- [| liftE . return . modeOp BFS . allValues . nf |]
flattener <- [| liftE . nf |]
```

We can now implement the complete scheme for generating a set function implementation as follows:

```
return $ genNDLam (VarP <$> (f : xs))
       $ genBinds (AppE flattener . VarE <$> xs) (VarP <$> xs')
       $ genBindStore
       $ AppE finder
       $ genPutStore
       $ genNDApply (VarE f) (VarE <$> xs')
```

As proposed in section 3.3.1, we want to provide a fixed number of set functions for use in Curry modules, which we can now do easily by defining `set0 = $(setN 0)`, `set1 = $(setN 1)` and so on.

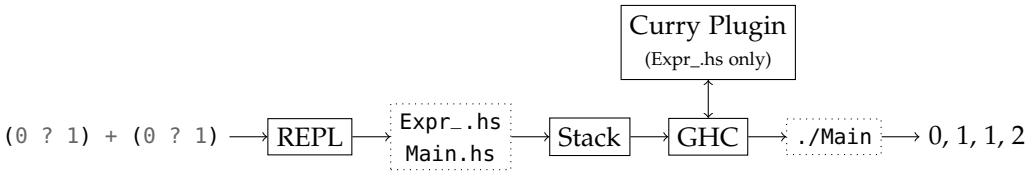


Figure 4.5. Interactive expression evaluation pipeline

4.4 Curry Plugin REPL

The final major component that we will implement is the REPL for Curry introduced in section 3.4. The REPL makes it easy for programmers to experiment with expressions on-the-fly and plays an important role in facilitating a quick feedback loop during development.

Due to the challenges involved in providing a fully-featured Curry REPL directly inside GHCi and the boilerplate required to wrap nondeterministic computations manually, as shown in section 3.4.1, we will take a different approach and implement the REPL as a separate program that is decoupled from the plugin itself. Since the main responsibility of the REPL lies in chaining GHC invocations together and generating code, a scripting language naturally fits our use case. For our implementation, we will use *Python*, mainly due to its extensive support for interactive command line interfaces and subprocess interfacing within the standard library.

4.4.1 High-Level Architecture

The REPL user interface is implemented using the `cmd` framework from the Python standard library, which provides a foundation for interactive command line interfaces and command processing. Under the hood, the REPL generates Haskell modules and calls into the Haskell Stack build tool, which is used both to build modules with the plugin and to build the plugin itself. When invoking GHC through Stack, the plugin is automatically provided as part of the package environment and can be activated through the corresponding GHC flag, which the REPL automatically sets for the generated expression module. An overview of the expression compilation and evaluation pipeline can be found in figure 4.5. In the following sections we will take a more detailed look at how the REPL's core features are implemented.

4.4.2 Expression Evaluation

In order to evaluate expressions, the REPL has to generate source files for compilation with the plugin. Following the precedent of other Curry compilers, the REPL uses a subdirectory of the `.curry` directory named `curry-plugin-repl-0.0.1` to place generated modules and compilation artifacts. For an expression the REPL produces two source files: First, it creates the module `Expr_.hs`[11] that binds the user's expression to a top-level function named `expr_` and uses the plugin's lifting, along with imports of previously loaded modules, custom flags and interactively added type declarations. Then, it generates a main function in `Main.hs` that encapsulates the computation and prints the results:

[11] The name of this module is arbitrary, but includes an underscore to make clashes with regular modules that follow Haskell's naming conventions unlikely.

```
main :: IO ()
main | null results = putStrLn "*** No value found!"
     | otherwise    = mapM_ print results
  where results = $(evalGeneric BFS '_expr)
```

We default to BFS for the same reasons as indicated in section 4.2.1. The search strategy, however, is customizable through the `:strategy` command. While the given implementation of `main` works well for simple examples, it includes a subtle assumption that turns out not to hold for every data type in the nondeterministic world: The resulting values need to have an instance of `Prelude.Show`, as required by `print`. For example, if we declare `data Peano = Zero | Succ Peano deriving Show` interactively and evaluate `Succ (Succ Zero)`, we get a compile-time error:

```
error:
  • No instance for (Show Peano) arising from a use of ‘print’
  • In the first argument of ‘mapM_’, namely ‘print’
    In the expression: mapM_ print results
   |
22 |      | otherwise    = mapM_ print results
   |                             ^^^^^
```

The issue, in this case, is that the plugin only synthesizes an instance of `ShowND` for the lifted type, not however `Show` for the unlifted type. Since we are primarily interested in showing values that originate from the nondeterministic world, we can replace the usage of `print` with a custom function that leverages the `ShowND` instance by lifting and unlifting the type again:

```
printWithShowND :: (NormalformN a, BuiltIn.ShowND a) => UnliftedN a -> IO ()
printWithShowND = putStrLn . head
                           . modeOp DFS
                           . allValues
                           . nf . (BuiltIn.show >>$) . liftE
                           . return
```

Using this function we can now show both built-in and user-defined data types in REPL.

Besides computing pure values, the user may also want to perform IO in the REPL though. Since IO actions require a fundamentally different way of evaluation compared to pure values, we need a mechanism to overload the evaluation function. Fortunately, Haskell's type classes are well-suited for this task. We begin by abstracting the print function into a class method:

```
class ReplEval a where
  replEval :: a -> IO ()
```

Now, we provide two primary instances, one for showable values and another one for IO values. The instance for showable values simply delegates to the `printWithShowND` function introduced previously:

```
instance (NormalformN a, BuiltIn.ShowND a, b ~ UnliftedN a) => ReplEval b where
  replEval = printWithShowND
```

The IO-based instance, on the other hand, binds the value first, thus potentially performing IO, and then prints the value:

```
instance (NormalformN a, BuiltIn.ShowND a, b ~ UnliftedN a) => ReplEval (IO b) where
  replEval x = x >>= printWithShowND
```

Even though IO values usually do not carry a `ShowND` instance, this cannot be guaranteed in general, therefore these instances are potentially overlapping. Since we want the IO-based evaluation to take precedence in the case that the value also carries a `ShowND` instance, we additionally annotate the first instance with an `{-# OVERLAPPABLE #-}` pragma. In addition to the instances given above, we also provide a way to 'display' functions in the REPL by printing a placeholder value, to avoid an error message in such cases:

```
instance ReplEval (a -> b) where
  replEval _ = putStrLn "<function>"
```

With these instances in place, the user can now seamlessly evaluate both pure and IO values in the same manner. Note that the value of an IO action is still always shown, however, even if it is unit. This means that evaluating `putStrLn "Hello world"` will output both `Hello world` and `()`.

4.4.3 Type Evaluation

Similar to GHCi, PAKCS and KiCS2, the Curry plugin REPL should have the ability to display the inferred type of an expression. As proposed in section 3.4.2, we will add the `:type` and `:mtype` commands for querying the unlifted and lifted type of an expression, respectively. Perhaps surprisingly, extracting the inferred type of a lifted binding directly from the compiler is not a trivial task. Neither parsing the type from `-ddump-tc` output nor introducing a custom flag in the plugin that outputs the right information feels like a satisfactory solution, the former additionally suffers from the inconsistencies where the lifted type would in some cases be reported instead of the unlifted type[12]. For this reason, we pick a different approach and extract the type information via Template Haskell[13].

Like in the previous section, we generate an `Expr_.hs` module. This time, however, we want to use Template Haskell to extract the type from the expression. Due to GHC's stage restriction, we need to separate our Template Haskell code from its splice-based usage at the module-level, therefore we move this logic to a new module named `Type_.hs`. Through Template Haskell's type reification mechanism, we fetch the lifted type of the expression at compile-time:

```
_type :: Q Type
_type = reifyType '_expr
```

Template Haskell's splices now let us use this type like any other type expression in Haskell. Depending on whether we are interested in the lifted or the unlifted type, we may have to do some preprocessing first, however. For this, we introduce a type family named `ReplType`. If we are interested in the lifted type, no further processing is required, therefore `ReplType` can be defined as the identity synonym:

```
type ReplType a = a
```

If, on the other hand, we are interested in the unlifted type, which is often the case, we need to remove the outer `Nondet` constructor and use `UnliftedN` to fetch the unlifted type. To do so, we pattern-match the `Nondet` constructor at the type level with a closed type family[14]:

```
type family ReplType a where
  ReplType (Nondet a) = UnliftedN a
```

[12] For example, compiling `null` yields `_expr :: Nondet (ListND a --> Bool)`, whereas `null [1, 2]` compiles to `_expr :: Bool`.

[13] Unfortunately, we cannot output polymorphic types with this approach yet, since binding a polymorphic function like `show` at the top-level of a Curry module without a type annotation yields an error reporting ambiguous types.

[14] This type family is both closed and non-exhaustive, a perhaps unusual combination. We expect it to always match the equation, however, as the lifted type of every nondeterministic expression has to be wrapped in `Nondet`.

```
> :t \x -> putStrLn (a : x)
\x -> putStrLn ('a' : x) :: [Char] -> IO ()

> :mt [1, 2, 3]
[1, 2, 3] :: Nondet (ListND Integer)
```

Figure 4.6. Inferred types in the REPL

Finally, we use `typeRep` from Haskell's `Type.Reflection` to pretty-print the type. To avoid the otherwise ambiguous type instantiation, we also need to use the `TypeApplications` language extension to explicitly supply a type, in our case the mapped `ReplType`, using the `@`-syntax. Putting it all together, our final main function that outputs the inferred type for a Curry expression, generated as part of `Main.hs` looks like this:

```
main :: IO ()
main = putStrLn ("<expression> :: " ++ show t)
  where t = typeRep @(ReplType $(_type))
```

Here, `<expression>` denotes the raw expression string as entered by the user, which will be included as a literal string during the generation of this module. A few examples of inferred types shown using the REPL can be found in figure 4.6.

4.4.4 Module Loading

To support multi-module Curry projects, our REPL will include the commands `:add`, `:load` and `:reload` for loading modules into the REPL context, as detailed in section 3.4.2. Since our REPL does not operate within a compiler context, we cannot load the compiled modules into memory directly like GHCi. GHC, however, provides support for module-level incremental compilation, which we use in conjunction with the `--make` flag to compile a loaded module along with its imports to object files. Once the user evaluates an expression, the loaded modules are added to the import list in `Expr_.hs`. The expression evaluation mechanism also uses the `--make` flag, therefore these modules are automatically included in the compilation and, if they have not changed in the meantime, reuse the object file generated during loading.

4.4.5 Miscellaneous

The list in section 3.4.2 mentions a few other commands, which are not central to the REPL, but provide convenient side functionality. This notably includes `:save`, a command that lets the user export the compiled expression evaluation to an executable. Since the mechanism detailed in section 4.4.2 already generates such an executable, we only need to copy it in a single additional step to the current directory in order to 'save' it.

For scriptability, the REPL will additionally accept arguments in the form of command invocations, making it possible to perform non-interactive expression evaluations, such as with `$REPL :eval 3 + 3 :q`, where `$REPL` is the path to the REPL.

Chapter 5

Evaluation

In this chapter we will evaluate the implemented features and approaches with regard to completeness, extensibility, maintainability, testability and performance.

5.1 Extensibility and Maintainability

Both the data type class and the set functions are implemented almost exclusively at the language level, with only the final anyclass deriving requiring a minor change to the plugin's derive transformation. This makes these features easy to reason about and to extend. Many features, notably some unification operators such as `(=:=)`, for example, can be defined in terms of the data type class as plain library functions and do not require deep support by the compiler itself. This is also advantageous from a maintainer's perspective, since they do not have to have special knowledge of the compiler's internals aside from the monadic representation of nondeterminism to work on these features. Another nice side effect of moving such features into the language is that they do not rely as much on internal GHC APIs, which may change rapidly across GHC releases.

The REPL, which is implemented as a completely separate component, exhibits the same advantages with regard to extensibility and maintainability. Being decoupled from the plugin's internals also makes it possible to implement meta-commands such as `:rebuild`, which let developers of the Curry plugin rebuild it without leaving the REPL, thereby facilitating a shorter development feedback loop.

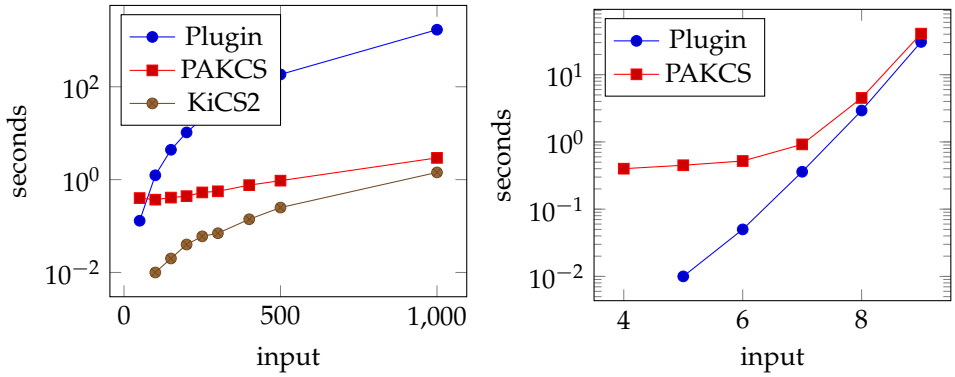
5.2 Testability

The REPL, due to its versatile and scriptable interface, is an excellent tool for automatically testing commands and compilation results. To verify that the Curry system handles a range of simple use cases correctly, we add a Bash script that performs a variety of smoke tests in the form of REPL invocations. These exercise central language features that we introduced in the previous sections, notably including simple expressions, both deterministic and nondeterministic, set functions, with and without sharing, free variables and IO. Additionally, they verify that modules, with and without imports, can be loaded and used in the REPL.

The test cases themselves are implemented as simple command invocations that use `grep` to check for the expected output and exit with a non-zero code if this fails. Since piped output is generally not visible to the user, we additionally T-pipe it to the console. An example can be found in figure 5.1.

```
echo "==> Testing basic set functions..."
$REPL :eval "set2 (?) 9 (3 ? 4)" :q | tee /dev/tty | grep -zqP "\[9,3\]\n\[9,4\]"
  || error "Basic set functions failed!"
```

Figure 5.1. Example of Curry REPL smoke test



(a) Free variables (Halving Peano numbers) **(b)** Set functions (N-Queens)

Figure 5.2. Run-time performance benchmarks

5.3 Features

Our implementation extends the Curry plugin with the majority of features that a Curry programmer comes to expect from a Curry system: Free variables and set functions improve expressivity, in particular for logic and constraint-solving programs, while the availability of IO in Curry modules paves the road to making the plugin-based Curry compiler a general-purpose programming system. Still, our plugin-based compiler has a major limitation in comparison with PAKCS and KiCS2, namely the lack of non-strict unification. Laziness embodies one of the core features of Curry and making unification more efficient through call-by-need is one of its central strengths when compared to other high-level implementations of nondeterminism. Integrating non-strict unification into the plugin, however, requires deep modifications to the monad that represents nondeterminism, which we will consider out of scope for this work. We will discuss some more details on this in section 6.2.3.

5.4 Performance

Finally, we will evaluate the run-time performance of the implemented features. Since our main focus in this work is correctness and general availability of these features, we do not expect them to outperform compilers with native support for features such as free variables and non-strict unification. Still, these benchmarks may provide useful insights into the performance characteristics of a language-level implementation of such features and a baseline for future work.

To evaluate the performance, we will use a slightly modified variant of the KiCS2 benchmark suite and test it against the plugin, KiCS2 and PAKCS.

5.4.1 Free Variables as Generators

Generators are a powerful abstraction that provide a nice theoretical framework for modeling free variables in functional logic programs. Our implementation of free variables using `aValue`, however, is noticeably slower than the equivalent program using native free variables in KiCS2 and PAKCS, as a benchmark performing arithmetic on Peano numbers shows in figure 5.2a.

One reason for this is that we represent free variables explicitly in the language and implement unification as an equality operator. While the strategy is semantically correct, it greatly increases the

search space of expressions when compared to a native implementation of unification in the compiler, leading to potentially quadratic run-time as in figure 5.2a. Additionally, our implementation is stricter than it ought to be. For example, evaluating `let x free in x :: [Bool]` in PAKCS or KiCS2 does not evaluate the `[Bool]`. Instead, it returns an anonymous variable representing an arbitrary `[Bool]`, whereas our plugin would list all `[Bool]`s.

5.4.2 Set Functions

Our implementation of set functions is similar to PAKCS in that the arguments passed to the encapsulated function are evaluated strictly. It should therefore not come as a surprise that our plugin performs similarly. A benchmark that solves the n-queens problem for various input sizes, as plotted in figure 5.2b, shows that the Curry plugin is actually slightly faster than PAKCS, mainly due to the lack of interpretation overhead[1]. Like with the implementation of non-strict unification, we will leave the implementation of non-strict set functions up for future research, as we will discuss in section 6.2.2.

5.4.3 REPL Compilation

The performance of the Curry plugin REPL is comparable to the KiCS2 REPL, which also evaluates expressions by generating a module and passing it to the compiler. While this mechanism causes the initial compilation of an expression to be slower than in interpreter-based REPLs such as PAKCS, the compiled program is completely native and therefore comparatively fast.

[1] We omit KiCS2 for now, since set functions do not compile with KiCS2 3.0.0 at the time of writing.

Chapter 6

Conclusion

In this final chapter we will summarize our implementation, briefly recapitulate the results and finally provide suggestions for future work, thereby concluding the thesis.

6.1 Summary and Results

As our experience shows, GHC plugins provide an excellent base for extending Haskell with new language features. The advantage of a plugin-based approach, especially when compared to the generation of Haskell code in compilers like KiCS2, is that we can reuse many parts of the compiler pipeline without the need for a custom frontend that is written from scratch. By taking advantage of the Curry plugin's groundwork, we can implement many features of existing contemporary Curry compilers on top of call-time-choice nondeterminism with little additional development overhead. Furthermore, GHC's extensive support for metaprogramming lets us move large parts of the implementation of free variables, unification and set functions directly into the language. Keeping these features largely decoupled from the plugin's internal transformations is especially advantageous from a standpoint of maintainability and extensibility, as GHC's internal APIs are known to change rapidly across releases, whereas language-level APIs like Template Haskell or Generics are stable and well-documented.

In addition to the aforementioned language features, we provide a REPL as an interactive interface to the compiler and the plugin. Using the REPL, users can evaluate and execute Curry programs as naturally as with PAKCS or KiCS2, without compromising on flexibility or run-time performance.

The only major limitation of our approach is the comparatively high strictness in the implemented features, as well as the lack of native unification and the resulting performance implications. We therefore provide suggestions for approaches on how to mitigate these restrictions in the final section.

6.2 Future Work

While our implementation includes most of Curry's characteristic features, there are still some improvements and generalizations to be left for future research, a few of which we will outline in the following.

6.2.1 Polyvariadic Set Function Operator

Set functions are currently only provided in a limited number of fixed arities as described in section 3.3.1. Having a truly polyvariadic set function operator would therefore be nice to have as it would allow us to replace `set1`, `set2`, etc. with a single `set` operator. This might be feasible to implement with the Curry plugin's support for multi-parameter type classes and functional dependencies as shown in figure 6.1.

```
class SetFunction f g | f -> g where
  set :: f -> g

instance SetFunction b [b]
instance SetFunction f g => SetFunction (a -> f) (a -> g)
```

Figure 6.1. Polyvariadic set function operator using a multi-parameter type class

```
type family Set f where
  Set (a -> b) = a -> Set b
  Set b        = [b]

class SetFunction f where
  set :: f -> Set f
```

Figure 6.2. Polyvariadic set function operator using a closed type family

There are, however, some challenges involved in avoiding overlap, particularly due to a lack of direct support for closed world instances in GHC. Closed type families could be used to model such a function too as shown in figure 6.2.

6.2.2 Non-Strict Set Functions

As discussed earlier, the set functions we implemented are strict in their arguments, which makes them more restrictive than their theoretical counterparts introduced in section 3.3.1. A truly lazy set function implementation would for example let us evaluate expressions such as `set2 const 1 failed`, which in our current implementation would fail instead of yielding `1`. Since such set functions require non-trivial changes to the plugin's internals, we leave this enhancement for future research.

6.2.3 Non-Strict Unification

Another open improvement that relates to laziness includes the implementation of a *non-strict unification operator* as indicated in section 5.3. This operator, also named `(=:<=)`, is a generalization of the strict unification operator `(=:=)` which is less strict in its right argument [AH06a]. For example, evaluating the following term would yield `True` instead of failing, as would be case with `(=:=)`:

```
x' =:<= failed
  where x' free
```

In contrast to `(=:=)`, `(=:<=)` is no longer symmetric and evaluates the right-hand side lazily as it tries to unify it with the left-hand side. Specifically, this unification can be viewed as an operation where the right-hand side (the 'value') is 'bound' to the left-hand side (the 'pattern').

The main challenge involved with non-strict unification is that it requires native support for free variables within the monadic representation of Curry values, whose implementation is quite a bit more complex than the language-level synthesis of `(===)` and `aValue` with `Data` presented in this work.

6.2.4 Functional Patterns

A common language extension of Curry systems are *functional patterns* as proposed in [AH06a], permitting declarations like the following:

```
last :: Data a => [a] -> a
last (_ ++ [x]) = x
```

These are especially nice, because they naturally extend Haskell's pattern syntax, allowing the use of arbitrary functions instead of just data constructors. We could use `(=:=)` to desugar this example:

```
last :: Data a => [a] -> a
last xs | _ ++ [x] =:= xs = x
  where x free
```

This, however, makes `last` more strict than it ought to be, as terms like `last [failed, 1]` would not evaluate to anything. Replacing `(=:=)` with the non-strict unification `(=:<=)` would provide us with the correct semantics, causing the example to evaluate to `1`.

Bibliography

[AH00] Sergio Antoy and Michael Hanus. "Compiling multi-paradigm declarative programs into Prolog". In: *Frontiers of Combining Systems*. Ed. by Hélène Kirchner and Christophe Ringeissen. Vol. 1794. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 171–185. DOI: 10.1007/10720084_12.

[AH06a] Sergio Antoy and Michael Hanus. "Declarative programming with function patterns". In: *Logic Based Program Synthesis and Transformation*. Ed. by Patricia M. Hill. Vol. 3901. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 6–22. DOI: 10.1007/11680093_2.

[AH06b] Sergio Antoy and Michael Hanus. "Overlapping rules and logic variables in functional logic programs". In: *Logic Programming*. Ed. by Sandro Etalle and Mirosław Truszczyński. Vol. 4079. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 87–101. DOI: 10.1007/11799573_9.

[AH09] Sergio Antoy and Michael Hanus. "Set functions for functional logic programming". In: *Proceedings of the 11th ACM SIGPLAN conference on Principles and practice of declarative programming - PPDP '09*. the 11th ACM SIGPLAN conference. Coimbra, Portugal: ACM Press, 2009, p. 73. DOI: 10.1145/1599410.1599420.

[BHH04] Bernd Braßel, Michael Hanus, and Frank Huch. "Encapsulating non-determinism in functional logic computations". In: *Journal of Functional and Logic Programming* (Jan. 2004). URL: https://www.informatik.uni-kiel.de/~mh/papers/JFLP04_findall.pdf.

[BHPR11] Bernd Braßel, Michael Hanus, Björn Peemöller, and Fabian Reck. "KiCS2: a new compiler from Curry to Haskell". In: *Functional and Constraint Logic Programming*. Ed. by Herbert Kuchen. Vol. 6816. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 1–18. DOI: 10.1007/978-3-642-22531-4_1.

[Fie14] Robert Field. *JEP 222: jshell: the Java shell (read-eval-print loop)*. May 16, 2014. URL: https://openjdk.java.net/jeps/222 (visited on 09/25/2021).

[FKS11] Sebastian Fischer, Oleg Kiselyov, and Chung-Chieh Shan. "Purely functional lazy nondeterministic programming". In: *Journal of Functional Programming* 21.4 (Sept. 2011), pp. 413–465. ISSN: 0956-7968, 1469-7653. DOI: 10.1017/S0956796811000189.

[HA77] M. C.B. Hennessy and E. A. Ashcroft. "Parameter-passing mechanisms and nondeterminism". In: *Proceedings of the ninth annual ACM symposium on Theory of computing - STOC '77*. the ninth annual ACM symposium. Boulder, Colorado, United States: ACM Press, 1977, pp. 306–311. DOI: 10.1145/800105.803420.

[Han] Michael Hanus. *Curry examples*. URL: https://www.informatik.uni-kiel.de/~mh/curry/examples/ (visited on 08/02/2021).

[Han16] Michael Hanus. *Curry: an integrated functional logic language*. Jan. 13, 2016. URL: https://www-ps.informatik.uni-kiel.de/currywiki/_media/documentation/report.pdf.

[Han18] Michael Hanus. *Setfunctions package for Curry*. GitLab. Dec. 31, 2018. URL: https://git.ps.informatik.uni-kiel.de/curry-packages/setfunctions (visited on 08/29/2021).

[Han19] Michael Hanus. "Declarative programming languages - lecture notes". 2019. URL: https://www-ps.informatik.uni-kiel.de/~mh/lehre/dps19/.

[Han99] Michael Hanus. "Distributed programming in a multi-paradigm declarative language". In: *Principles and Practice of Declarative Programming*. Ed. by Gopalan Nadathur. Vol. 1702. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 188–205. DOI: 10.1007/10704567_11.

[HK08] Michael Hanus and Christof Kluß. "Declarative programming of user interfaces". In: *Practical Aspects of Declarative Languages*. Ed. by Andy Gill and Terrance Swift. Vol. 5418. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 16–30. DOI: 10.1007/978-3-540-92995-6_2.

[HK10] Michael Hanus and Sven Koschnicke. "An ER-based framework for declarative web programming". In: *PADL'10: Proceedings of the 12th international conference on Practical Aspects of Declarative Languages*. Jan. 2010, pp. 201–216. DOI: 10.1007/978-3-642-11503-5_18.

[HT20a] Michael Hanus and Finn Teegen. "Adding data to Curry". In: *Declarative Programming And Knowledge Management* 12057 (May 5, 2020), pp. 230–246. DOI: 10.1007/978-3-030-46714-2_15.

[HT20b] Michael Hanus and Finn Teegen. "Memoized pull-tabbing for functional logic programming". In: *arXiv:2008.11999 [cs]* (Aug. 27, 2020). arXiv: 2008.11999. URL: http://arxiv.org/abs/2008.11999.

[Kis] Oleg Kiselyov. *Polyvariadic functions*. URL: http://okmij.org/ftp/Haskell/polyvariadic.html (visited on 08/15/2021).

[MDJL10] José Pedro Magalhães, Atze Dijkstra, Johan Jeuring, and Andres Löh. "A generic deriving mechanism for Haskell". In: *ACM SIGPLAN Notices* 45.11 (Nov. 2010), pp. 37–48. DOI: 10.1145/2088456.1863529.

[MPJ12] Simon Marlow and Simon Peyton-Jones. "The Glasgow Haskell Compiler". In: *The Architecture of Open Source Applications*. Vol. 2. Lulu, Mar. 16, 2012. URL: https://www.microsoft.com/en-us/research/wp-content/uploads/2012/01/aos.pdf.

[NPJ17] Shayan Najd and Simon Peyton-Jones. "Trees that grow". In: *Journal of Universal Computer Science* 23 (Jan. 28, 2017). DOI: 10.3217/JUCS-023-01-0042.

[Pro20] Kai-Oliver Prott. "Extending the Glasgow Haskell Compiler for functional-logic programs with Curry-Plugin". Master's Thesis. Kiel University, Oct. 2020. URL: https://www.informatik.uni-kiel.de/~mh/lehre/abschlussarbeiten/msc/Prott.pdf.

[PWN19] Matthew Pickering, Nicolas Wu, and Boldizsár Németh. "Working with source plugins". In: *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell - Haskell 2019*. the 12th ACM SIGPLAN International Symposium. Berlin, Germany: ACM Press, 2019, pp. 85–97. DOI: 10.1145/3331545.3342599.

[SJSC08] Tom Schrijvers, Simon Peyton Jones, Martin Sulzmann, and Manuel Chakravarty. "Type checking with open type functions". In: *ACM SIGPLAN Notices* 43.9 (Sept. 2008), pp. 51–62. DOI: 10.1145/1411203.1411215.

[Tea20] GHC Team. *Extending and using GHC as a library*. Glasgow Haskell Compiler 9.0.1 User's Guide. 2020. URL: https://downloads.haskell.org/~ghc/latest/docs/html/users_guide/extending_ghc.html (visited on 08/05/2021).

[Wad90] Philip Wadler. "Comprehending monads". In: *Proceedings of the 1990 ACM conference on LISP and functional programming - LFP '90*. the 1990 ACM conference. Nice, France: ACM Press, 1990, pp. 61–78. DOI: 10.1145/91556.91592.

Acronyms

API Application Programming Interface. 1–3, 6, 35, 39

AST Abstract Syntax Tree. 1, 6, 7, 29

BFS Breadth-First Search. 1, 15, 24, 27, 32

DFS Depth-First Search. 1, 15, 24

GHC Glasgow Haskell Compiler. v, 1, 2, 5–7, 9, 13, 15, 17, 18, 23, 26, 31, 33–35, 39, 40

GHCi GHC's interactive shell. 1, 13–15, 31, 33, 34

IO Input/Output. v, 1, 9–11, 15, 23–27, 32, 33, 35, 36, 47

KiCS2 Kiel Curry System Version 2. 1, 5, 9, 11, 13–15, 33, 36, 37, 39

PAKCS Portland Aachen Kiel Curry System. 1, 5, 9, 11, 13–15, 27, 28, 33, 36, 37, 39

REPL Read-Eval-Print-Loop. v, 1, 2, 5, 9, 13–15, 17, 23, 31–35, 37, 39

Appendix A

Complete Example of a Curry Module

The following is a complete example of a Curry module showcasing the new features, including free variables, unification, set functions and IO:

```
{-# OPTIONS_GHC -fplugin Plugin.CurryPlugin #-}
module Example where

data Peano = O | S Peano deriving Show

toPeano :: Int -> Peano
toPeano n | n == 0    = O
          | otherwise = S (toPeano (n-1))

fromPeano :: Peano -> Int
fromPeano O     = 0
fromPeano (S p) = fromPeano p + 1

add :: Peano -> Peano -> Peano
add O     p = p
add (S p) q = S (add p q)

half :: Peano -> Peano
half y | (add x x) =:= y = x
  where x = aValue :: Peano

infixr <>
(<>) :: [a] -> [a] -> [a]
[]     <> ys = ys
(x:xs) <> ys = x : xs <> ys

permute :: [a] -> [a]
permute []     = []
permute (x:xs) = insX (permute xs)
  where insX []     = [x]
        insX (y:ys) = x : y : ys ? y : insX ys

permute' :: [a] -> [[a]]
permute' = set1 permute
```

```
fortyTwo :: String
fortyTwo = show (fromPeano (half (toPeano 84)))

main :: IO ()
main = do
  let perms = permute' fortyTwo
  putStrLn $ "The answer to life, the universe and everything might be " <> last perms
  putStrLn $ "It is " <> fortyTwo <> " though"
```