Programming Languages and Compiler Construction
Department of Computer Science
Kiel University

**Master's Thesis**

# Extending the Glasgow Haskell Compiler for functional-logic Programs with *Curry-Plugin*

Kai-Oliver Prott

October 2020

Supervisors:
Prof. Dr. Michael Hanus
M.Sc. Finn Teegen

# Erklärung der Urheberschaft

Hiermit versichere ich, dass ich die von mir vorgelegte Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form in keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

| Ort, Datum | Unterschrift |
|---|---|

# Abstract

Curry is a functional logic programming language with Haskell-like syntax and support for nondeterministic computations with free variables. In recent years, the language has been extended to support type classes and higher-rank types. But when compared with Haskell, Curry is still missing a lot of useful language extensions. Instead of implementing all those extensions in the existing Curry compiler, the idea is to use the Glasgow Haskell Compiler plugin infrastructure for a new compiler.

The result of this effort is *Curry-Plugin*, a GHC plugin that supports Curry-style nondeterminism with call-time choice in a Haskell module. The *Curry-Plugin* performs a monadic lifting of all datatypes and functions in a module and then instantiates that monad with a type that provides the desired nondeterministic semantics with call-time choice. It also provides the operator `(?)`, which nondeterministically chooses between its arguments. Due to the usage of GHC, a lot of Haskell's language extensions can be supported "for free", while some of them are incompatible with the monadic lifting used by this plugin.

# Contents

**1. Introduction** **1**
1.1. Contributions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1
1.2. Outline . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2

**2. Preliminaries** **3**
2.1. Curry . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
2.1.1. Nondeterminism . . . . . . . . . . . . . . . . . . . . . . . . . . 3
2.1.2. Call-Time Choice and Run-Time Choice . . . . . . . . . . . . . . 5
2.1.3. Free Variables . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
2.2. GHC's API and Plugin Infrastructure . . . . . . . . . . . . . . . . . . 6
2.2.1. Renamer Plugins . . . . . . . . . . . . . . . . . . . . . . . . . . 7
2.2.2. Type Checker Plugins . . . . . . . . . . . . . . . . . . . . . . . 7
2.2.3. Constraint Solver Plugins . . . . . . . . . . . . . . . . . . . . . 10

**3. Design** **13**
3.1. Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
3.1.1. Pattern Match Semantics . . . . . . . . . . . . . . . . . . . . . . 13
3.1.2. Encapsulating Nondeterministic Computations . . . . . . . . . . 15
3.1.3. Other Features and Restrictions . . . . . . . . . . . . . . . . . . 16
3.2. Using other Effects . . . . . . . . . . . . . . . . . . . . . . . . . . . 17

**4. Implementation** **19**
4.1. Monadic Lifting . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20
4.1.1. Lifting Data Types . . . . . . . . . . . . . . . . . . . . . . . . . 22
4.1.2. Lifting Functions . . . . . . . . . . . . . . . . . . . . . . . . . . 24
4.2. Pattern Matching . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
4.2.1. Translating Pattern Matching . . . . . . . . . . . . . . . . . . . 30
4.2.2. Translating Guards . . . . . . . . . . . . . . . . . . . . . . . . . 33
4.2.3. Translating Pattern Bindings . . . . . . . . . . . . . . . . . . . . 34
4.2.4. Translating Do-Notation . . . . . . . . . . . . . . . . . . . . . . 36
4.3. Sharing Effects . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37
4.3.1. A Type Class for Sharing . . . . . . . . . . . . . . . . . . . . . . 38
4.3.2. Inferring Shareable Constraints in a Polymorphic Context . . . . 39
4.3.3. Using Quantified Constraints for Polymorphic Sharing . . . . . . 40
4.3.4. Problems with Recursion and Explicit Sharing . . . . . . . . . . 42
4.4. Type Classes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43
4.4.1. Lifting Type Classes . . . . . . . . . . . . . . . . . . . . . . . . 43

4.4.2. Lifting Instances . . . . . 45
4.5. Built-In Type Definitions . . . . . 45
4.6. Importing modules . . . . . 47
4.6.1. Subverting GHC's Type Checker for Imports . . . . . 47
4.6.2. Marking Plugin Modules . . . . . 52
4.7. Further Implementation Details . . . . . 53

**5. Evaluation** **57**
5.1. Compilation Performance . . . . . 57
5.2. Execution Performance . . . . . 57
5.3. Language Features . . . . . 59
5.4. Maintainability . . . . . 59

**6. Conclusion** **61**
6.1. Summary and Results . . . . . 61
6.2. Related Work . . . . . 61
6.3. Future Work . . . . . 62
6.3.1. Improving the Transformation . . . . . 62
6.3.2. Language Extensions . . . . . 63
6.3.3. Generalization for other Effects . . . . . 64

**Bibliography** **67**

**A. Transformation of a Small Example** **71**

**B. Generic Implementation of Shareable** **73**

**C. List of Language Extension Support** **75**

# List of Figures

2.1. Schematic evaluation with call-time and run-time choice . . . . . . . . . 5
2.2. GHC compilation pipeline with plugin extension points . . . . . . . . . 7
2.3. GHC's internal type syntax . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

3.1. Nondeterministic computation of all permutations for a given list . . . . 14
3.2. Capturing a nondeterministic computation . . . . . . . . . . . . . . . . . 15
3.3. Using datatypes and type classes . . . . . . . . . . . . . . . . . . . . . . 17

4.1. Extension points used by the *Curry-Plugin* . . . . . . . . . . . . . . . . 19
4.2. Examples for the lifting of types . . . . . . . . . . . . . . . . . . . . . . 20
4.3. Lifting rules for types . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
4.4. Transformation rule for data type definitions . . . . . . . . . . . . . . . 22
4.5. Examples for the lifting of `data` types . . . . . . . . . . . . . . . . . . . 23
4.6. Examples for the lifting of `newtypes` and type synonyms . . . . . . . . . 24
4.7. Transformation rule for variables . . . . . . . . . . . . . . . . . . . . . . 26
4.8. Transformation rule for lambda abstractions . . . . . . . . . . . . . . . . 26
4.9. Transformation rule for applications . . . . . . . . . . . . . . . . . . . . 27
4.10. Transformation rules for data- and newtype constructors . . . . . . . . . 27
4.11. Transformation rules for case expressions and branches . . . . . . . . . . 28
4.12. Transformation rule for let expressions . . . . . . . . . . . . . . . . . . . 29
4.13. Nondeterministic rules to generate pattern variants . . . . . . . . . . . . 35
4.14. Lifting of class dictionary declarations . . . . . . . . . . . . . . . . . . . 44
4.15. Transforming an equality relation . . . . . . . . . . . . . . . . . . . . . . 49
4.16. Plugin-specific debug options . . . . . . . . . . . . . . . . . . . . . . . . 54

5.1. Execution time comparisons of both compilers and the plugin . . . . . . 58

# 1. Introduction

Curry [Hanus (Ed.), 2016] is a functional logic programming language with Haskell-like syntax and support for nondeterministic computations with free variables. In recent years, the language has been extended by Teegen [2016] to support type classes and by Matthes [2019] to allow higher-rank types. While there are still a lot of interesting language extensions in Haskell that could also be implemented for Curry, implementing those extensions requires a lot of effort and most of them will never find their way into current compilers. This is also true for many other "industrial" programming languages. Creating a small compiler for the core of a language might be relatively easy, but making the language stable and feature-rich takes time. For example, the language *Rust* began 2006 as a personal project of a Mozilla employee and even after officially announcing the language in 2010, it took another five years before Rust saw its first stable release [The Rust Team, 2016].

Most compilers, especially the GCC for all its C-like languages, only re-use parts of the machine code generation across multiple supported languages and compilers by transforming all languages to a common intermediate language and then using a common backend for the rest of the compilation of all languages. Using the same backend across multiple languages allows the compiler developers to write code optimizations and machine-code generation just once without any apparent drawbacks.

A lot of existing compilers also allow third-party code to modify their compilation pipeline with a plugin. Examples for this include the GCC and Glasgow Haskell compiler [Pickering, Wu, and Németh, 2019]. While plugins are mostly used to implement small tools for a language by reading information generated during the compilation, a plugin can also be used to modify code between compilation phases.

Instead of extending current Curry compilers with new features, this thesis aims to create an entirely new compiler by utilizing the existing infrastructure of the Glasgow Haskell Compiler (GHC). By implementing a *Curry-Plugin* using the plugin infrastructure of GHC, we can leverage the Haskell compiler for most of the "heavy-lifting" during compilation, like type checking, optimizations and code generation, while gaining some of the language extensions "for free".

## 1.1. Contributions

While the general feasibility of our approach to implement a compiler as a GHC plugin has been tested in a previous research project, this thesis aims to extend the existing prototype to support at least the full syntax from Haskell's 2010 standard. Our prototype already contains a semantic transformation for simple programs. Extending the implementation will require solutions for sharing polymorphic variables, a correct

transformation of type classes and a special handling for imported definitions. While we could build a compiler with this approach, our aim is instead to make it possible to selectively transform only parts of a Haskell program. By adding functions to capture a nondeterministic computation in a Haskell module, we will make a seamless integration between Curry and Haskell code possible.

## 1.2. Outline

This thesis is composed of six main parts, including this introduction.

The second chapter establishes some preliminaries. It contains an introduction to the functional-logic programming language *Curry* (Section 2.1) and the Glasgow Haskell Compiler Plugin API (Section 2.2).

Chapter three gives an overview on the design of *Curry-Plugin* and how it is intended to be used (Section 3.1). The chapter also contains a section about modifying the plugin to support monadic effects other than nondeterminism (Section 3.2).

This is followed by the main chapter of this thesis, which describes the implementation of *Curry-Plugin*. It starts with a description of the monadic lifting used throughout the plugin (Section 4.1) and continues with a look at the pattern match translation (Section 4.2). The next section discusses everything required to share computations within the plugin (Section 4.3). Afterwards, we discuss how type classes can be lifted (Section 4.4) and how GHC's various built-in type constructors and type classes can be used and derived in the plugin (Section 4.5). In the next section, we take a look at how definitions can be imported across modules by subverting GHC's type checker (Section 4.6) and in the last section of the implementation chapter, we outline some smaller implementation details that were not explained earlier (Section 4.7).

Chapter five contains comparisons of the project complexity and performance (code size, speed) with other Curry compilers.

In the last chapter we summarize our results (Section 6.1), discuss some related work (Section 6.2) and conclude with an outlook on future work in Section 6.3.

# 2. Preliminaries

In this chapter, we talk about some preliminaries that are required to understand the design and implementation of the *Curry-Plugin*. We will introduce the programming language *Curry*, whose semantic model serves as a goal that we want to reach with the transformation implemented by our plugin. Our starting point for that transformation is GHC's internal representation of Haskell's Syntax, which we will explain alongside the plugin API of the compiler.

## 2.1. Curry

Curry [Hanus (Ed.), 2016] is a functional logic programming language with Haskell-like syntax and support for nondeterministic computations with free variables. We assume that the reader is familiar with Haskell's syntax and semantics and focus on the logic aspects of the Curry language instead, which are *nondeterminism*, *call-time choice* and *free variables*.

### 2.1.1. Nondeterminism

The value of an expression in Curry can be nondeterministic. A simple example of this is the nullary function `coin`, which simulates a coin flip and can either be of value `True` or `False`.

```
coin :: Bool
coin = True ? False
```

The so called *choice* operator `(?) :: a -> a -> a` chooses nondeterministically between its left and right argument and can occur nested inside an arbitrary expression. This operator serves as one of Curry's primitives to introduce nondeterminism to a computation.

**Example 2.1.1.** *Using the choice operator, we can define a function to nondeterministically insert an element into any position of a given list.*

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

*Our new function can be used to implement another function* `permutations`*, which nondeterministically computes any permutation of its input list.*

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

One thing to note about these function definitions are their type signatures: The type of their results does not indicate that the functions could be nondeterministic. This is, of course, intentional. It gives the language more flexibility and makes it easier to combine deterministic and nondeterministic definitions. A computational effect (like nondeterminism), that is not visible on the type level is called an *ambient* effect. Even Haskell is not a completely pure language. It provides *partiality* and *exceptions* as an ambient effect[1].

While the choice operator `(?)` is certainly useful for defining nondeterministic computations, it is actually implemented using Curry's other mechanism for introducing nondeterminism: Overlapping function rules. If the left side of two or more rules in a function definition overlap, all matching rules will be applied nondeterministically in Curry. This allows us to define the choice operator using two overlapping rules.

```
(?) :: a → a → a
x ? _ = x
_ ? y = y
```

Even though overlapping rules are arguably the more primitive way to introduce nondeterminism in Curry, the two concepts are equally powerful. Both of them can be expressed by using the other one.

If two or more overlapping rules can be expressed using the choice operator, then what happens in Curry if none of the rules match? We can try this by defining a function with one rule that never matches by using the guard syntax known from Haskell[2].

```
failed :: a
failed _ | False = undefined
```

In Haskell, evaluating the expression `failed :: Int` would lead to an exception that complains about the non-exhaustive patterns in our definition. If we evaluate the same expression in Curry, we end up with zero nondeterministic results. In fact, our definition of `failed` behaves like a neutral element to our choice function: For every expression `e`, the semantics of `(e ? failed)`, `(failed ? e)` and `(e)` are the same. The rest of this thesis will focus solely on the choice and failure operators for nondeterminism, the reasons will become clear when we talk about the design of the *Curry-Plugin* in Chapter 3.

**Nondeterminism as a Tree** An intuitive mental model for Curry's nondeterminism is the interpretation as a tree structure. A node in the tree corresponds to a choice between the two arguments of `(?)` with the leaves of the tree corresponding to nondeterministic results. The REPL (read/eval/print loop) of the KICS2 compiler for Curry allows us to enable `:set choices` to present all values as trees instead of term as shown below. We use the symbol `!` to mark failed branches of our computation.

[1]Using *unsafePerformIO*, we can even use I/O as some kind of ambient effect in GHC.
[2]While the function `failed` could be defined in plain Curry, it is actually provided as a primitive definition for performance reasons.

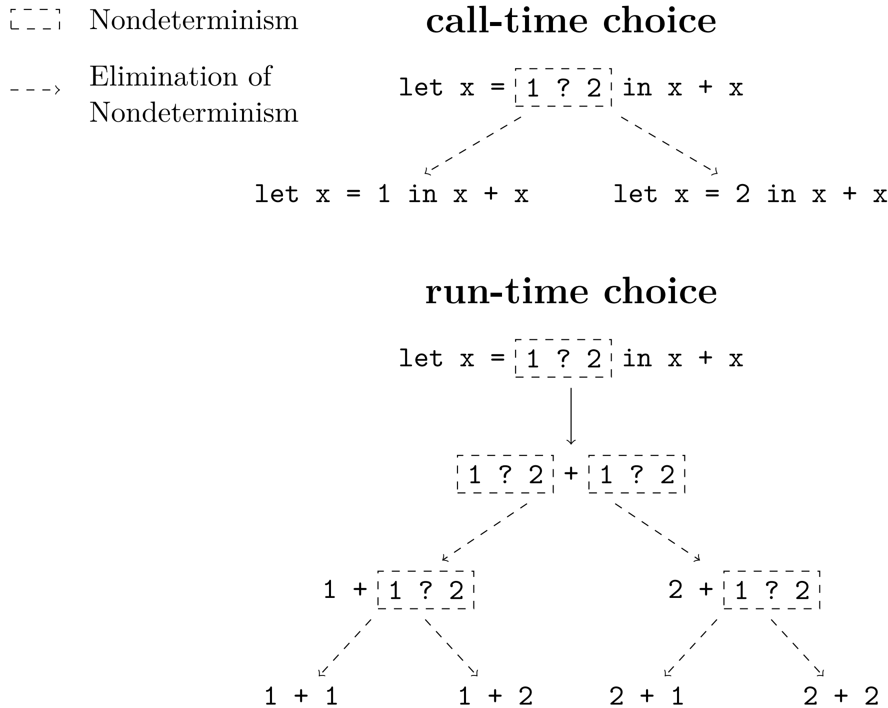


**Figure 2.1.:** Schematic evaluation with call-time and run-time choice

```
kics2> :set choices
kics2> (True ? False) : ([] ? failed)
?
|-- L: ?
|       |- L: [True]
|       --- R: !
--- R: ?
        |-- L: [False]
        --- R: !
```

As a next step, we will take a look at what happens if we combine Curry's nondeterminism effect with the laziness and *call-by-need* evaluation strategy known from Haskell.

### 2.1.2. Call-Time Choice and Run-Time Choice

If a variable is used multiple times in a Haskell or Curry expression, it is still only evaluated once due to the lazy *call-by-need* evaluation strategy used by both languages. We can say that the value is *shared* across all its use sites. This effect can be observed using unsafe language features. Evaluating the expression

```
let x = trace "Hello" 1 :: Int in x + x
```

in an interactive environment for Haskell or Curry will print "Hello" just once. If we

were to inline the definition of `x`, we would end up with two outputs of the same string. Like nondeterminism in Curry, tracing can be seen as an ambient effect in Haskell. So what happens if we replace the tracing from the example with a nondeterministic choice? If we evaluate

```
let x = 1 ? 2 :: Int in x + x
```

in Curry, we end up with two results: 2 and 4. The choice made in one of the occurrences of our variable `x` is shared across all its use sites. This interaction between laziness and nondeterminism is called *call-time choice* by Hennessy and Ashcroft [1977]. Instead of deciding on the value of a variable at call-time, one could also wait until its evaluation to fix the nondeterministic choice which is known as *run-time choice*. It is equivalent to in-lining the nondeterministic choice in `x` to its use-sites. Like in our tracing example, it would cause the semantics of our expression to change. Evaluating an inline variant of our previous example yields the results 2, 3, 3 and 4. Figure 2.1 shows a schematic evaluation with both call-time and run-time choice.

This interaction between lazy evaluation and ambient effects is not reserved to nondeterminism. In probabilistic programming for example, a similar concept to call-time choice is known as memoization [Cassel, 2014].

The last logic concept of Curry that we want to discuss are *free variables*.

### 2.1.3. Free Variables

Free variables can be defined in Curry using the keyword `free`. These variables can be used to compute unknown values. This concept allows us to define a function that returns the last element of a list using only one rule.

```
last :: Data a ⇒ a → a
last xs | ys ++ [e] == xs = e
  where ys, e free
```

The type class `Data` is a recent addition introduced by Hanus and Teegen [2020] to restrict the types on which the keyword `free` can be used. Functions, for example, have no instance of `Data`, because computing all suitable functions for a free variable of type `a -> b` is undecidable in general.

The class `Data` provides the function `aValue :: Data a => a`, which can be used to eliminate all occurrences of the `free` keyword from a program.

```
last :: Data a ⇒ a → a
last xs | ys ++ [e] == xs = e
  where { ys = aValue; e = aValue }
```

In the next section, we will take a look at some of GHC's internals and its plugin API.

## 2.2. GHC's API and Plugin Infrastructure

Like a lot of modern compilers, GHC's compilation pipeline can be augmented by the use of its plugin API [Pickering, Wu, and Németh, 2019]. There are several stages

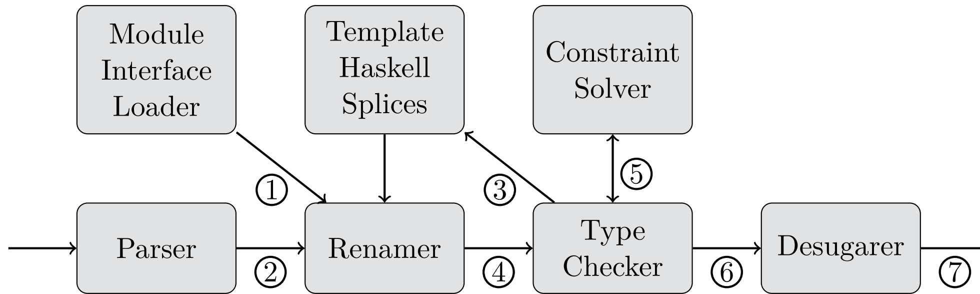


**Figure 2.2.:** GHC compilation pipeline with plugin extension points

in the compiler where a plugin can intercept the compilation to collect or modify GHC's internal knowledge about the module being compiled. These stages are shown in Figure 2.2. For this thesis, only the renamer (No. 4), typechecker (No. 6) and constraint solver (No. 5) extension points are relevant.

### 2.2.1. Renamer Plugins

After GHC's renaming pass, all identifiers have been disambiguated and are annotated with the module they originated from. Any plugin that is run at this point has access to the list of imported modules, their interfaces and all declaration groups (i.e. groups of mutually dependent top-level definitions), among other information.

GHC's abstract syntax tree is parameterized over the current compiler phase to form a tree with a growing amount of information [Najd and S. P. Jones, 2017]. Together with some type families, this mechanism assures that the same data types can be used throughout the main three phases (parsing, renaming, type checking) while still being able to annotate them with the information gathered during each phase. For example, in each position where the AST expects an identifier, a type family `IdP p` is used to map the current phase `p` to either `RdrName` (after parsing), `Name` (after renaming) or `Id` (after type checking).

### 2.2.2. Type Checker Plugins

After type checking, the syntax tree is annotated with accurate type information. At this stage, all instance definitions (either derived or user-defined) have been desugared into their dictionary bindings (see [M. P. Jones, 1995a]). This is the most useful phase for the *Curry-Plugin*, because the plugin requires accurate type information for its code transformation. While GHC actually has two representations for types — one user-facing and one internal representation — the only relevant type information after the type check is in the internal representation. When pretty printing the type information, GHC actually converts[3] its internal types to the more readable, user-facing format[4].

[3] This conversion can be suppressed with a compiler flag.

[4] It actually converts the types into the interface representation of types, but both look the same after pretty printing.

| | | | |
|---|---|---|---|
| *Type* | := | *Var* | (Type variable/constructor) |
| | \| | *Type Type* | (Type application) |
| | \| | `forall` *Var*. *Type* | (Invisible quantification) |
| | \| | `forall` *Var* → *Type* | (Visible quantification) |
| | \| | *Type* ⇒ *Type* | (Invisible function type) |
| | \| | *Type* → *Type* | (Visible function type) |
| | \| | *Literal* | (Type literal) |
| | \| | *Kind* `` `cast` `` *Coercion* | (Kind cast) |
| | \| | *Coercion* | (Coercion injected in type) |
| *Kind* | := | *Type* | (Kinds are types) |
| *Var* | := | Kind-annotated type variable | |
| | \| | Kind-annotated type constructor | |
| | \| | Type-annotated value constructor | (Promoted data type) |
| *Literal* | := | (Not relevant) | |
| *Coercion* | := | (Not relevant) | |

**Figure 2.3.:** GHC's internal type syntax

The internal type syntax is given in Figure 2.3. Note that the syntax contains a lot of constructs that are required for GHC's advanced language extensions, but are irrelevant for our plugin. We are interested in the internal representation of user-written type signatures from the 2010 Haskell standard only. One notable feature of GHC's internal representation is that it differentiates between visible (→) and invisible (⇒) function types. The latter are used to represent type class constraints, because GHC will later pass "invisible" arguments to functions that contain any constraints. The following example shows such a representation of two ordinary type signatures.

**Example 2.2.1.** *The type of* `fromIntegral` *is represented internally with two forall quantifiers and one invisible function type for each of the two constraints.*

```
-- | An ordinary function type signature
fromIntegral :: (Integral a, Num b) ⇒ a → b
-- Internal type: forall a. forall b. Integral a ⇒ Num b ⇒ a → b
```

*While the internal representation normally expects all variables quantified before any constraint is mentioned, type class functions use a different ordering where class constraints are written before other variables get quantified. This can be seen with the following function* `pure` *from the* `Applicative` *type class.*

```
-- | A function from a type class
pure :: Applicative f ⇒ a → f a
-- Internal type: forall f. Applicative f ⇒ forall a. a → f a
```

The type checked AST also contains explicit application of types and "evidence" to polymorphic functions in form of a wrapper expression. The wrapper is normally a *GHC Core* expression that is used to implement a corresponding constraint from the

function it is applied to. For a type class constraint, this wrapper contains a variable that is either implicitly bound in the function definition or it points directly to the dictionary implementing that type class.

**Example 2.2.2.** *If we look at an applicative version of the function* `replicate`*, we can see that it mentions an* `Applicative` *constraint in its type signature.*

```
replicateA :: Applicative m ⇒ Int → m a → m [a]
replicateA 0 _   = return []
replicateA n act = liftA2 (:) act (replicateA (n-1) act)
```

*The following code will be generated after inserting the explicit applications.*

```
replicateA :: Applicative m ⇒ Int → m a → m [a]
replicateA 0 _   = (return @ m @ dApplM @ [a]) ([] @ a)
replicateA n act = (liftA2 @ m @ dApplM @ a @ [a] @ [a])
                     ((:) @ a)
                     act
                     ((replicateA @ m @ dAppl @ a)
                        (((-) @ Int @ dNumInt) n 1)
                        act)
```

In Example 2.2.2, `dNumInt` is a dictionary for `Num Int` and `dApplM` is the dictionary provided to satisfy the `Applicative m` context. The former is created by the `Num` instance for `Int` in `GHC.Num`, while the latter will be passed to the function by the caller of `replicateA` as an invisible argument.

To modify or create new syntax, a plugin can either construct the required terms by directly creating the correct syntax tree datatypes, or it can use Template Haskell to let GHC create and type check the new syntax. It is hard to construct any required evidence in the former direct creation by hand, whereas the latter is easy to use on a small scale. However, Template Haskell can only be used to construct new syntax from already known pieces, which are in the scope of the Plugin source code. Using Template Haskell to safely update a function with all its type and evidence applications is not possible. We only use it to create the internal representation of monadic functions and other small code snippets which are required throughout our plugin.

**Example 2.2.3.** *The following code creates and type checks the AST representation for a* `return` *of type* `a → m a`

```
createReturn :: Type → Type → TcM (LHsExpr GhcTc)
createReturn a m =
  thExpr ← liftIO $ runQ [| return |]      -- Create ThExpr via quotation
  let expType = mkVisFunTy etype $  mkAppTy mty etype
  case convertToHsExpr (...) thExpr of     -- Convert ThExpr to HsExpr
    Left  msg → printBad msg
    Right res → do
      (rnExp, _)  ← (rnLExpr res)          -- Resolve names
      tcMonoExpr rnExpr (Check expType)    -- Check type
```

*Note that this code will use* `return` *from the scope at the definition of* `createReturn`*.*

Whenever we need to manually construct some part of a syntax tree, we can ask GHC to automatically solve any required constraints by communicating with GHC's constraint solver. The plugin only has to supply information about any given constraints that are currently in scope. These constraints are usually given by the type of the current binding we are manipulating. As a result of letting GHC solve all constraints, a plugin receives the evidence expressions that can be used to satisfy the required constraints. Combined with the Template Haskell approach, we can now efficiently and safely modify the full abstract syntax tree used throughout the compilation.

### 2.2.3. Constraint Solver Plugins

During type checking, GHC often needs to solve type class or other constraints to check if they are satisfiable in the checked function. The type checker might even create an equality constraint between two types that are supposed to be equal, if it cannot immediately decide if the two types can be unified. All of those constraints are sent to the constraint solver to check if they are valid. A constraint solver plugin can intercept these constraints at two different stages: After simplification of given constraints and after unflattening of wanted constraints. While the simplification and unflattening are part of GHC's constraint solver and not of interest to us, our plugin will later need to intercept or modify the given and wanted constraints after each of those phases. In general, a constraint solver plugin can inspect all constraints at the current stage and either say that it found a contradiction between constraints, a proof for a constraint, or it can tell GHC to extend its current list of constraints.

To tell GHC's constraint solver to get rid of a wanted constraint, a plugin needs to provide some kind of proof (e.g., a type class instance) and the evidence expression (e.g., a type class dictionary) to be used. Given constraints never require any evidence to drop, instead a given constraint contains the evidence for the correctness of its constraint already. For all new constraints, GHC will attempt to solve them before consulting the plugin again. This requires the author of a constraint solver plugin to be careful to not create any endless loops during a repeated analysis and transformation of constraints.

While all types of constraints can occur as given and wanted constraints, we will only focus on the parts that both have in common. For our purposes, a constraint is either a given/wanted type class constraint, or a wanted equality relation.

**Type Class Constraints** A type class constraint is annotated with the class that it mentions, as well as the type arguments that were used to instantiate any class variables. It also contains the type of the full constraint, which is just the class type constructor applied to all type arguments. A constraint `Monad []` will have the type `Monad []`, class `Monad` and the single argument `[]`. Wanted type class constraints always need to be supplied with the instance and dictionary term to solve them, whereas a given class constraint contains a reference to such evidence already.

**Equality Constraints** An equality relation will normally be marked by GHC as being irreducible, either because the compiler has not enough information to prove it, or because the compiler knows it to be false. The type of an equality constraint will be the equality type constructor `(~#)` applied to the kinds and then the types of both sides of the equality. If GHC tries to establish an equality between `Int` and `[]` for example, the constraint type would be: `(~#) Type (Type -> Type) Int []`. GHC will often omit the kinds and write the equality type constructor as infix, so we will do the same. The following ill-typed example shows which kinds of constraints get sent while type checking the code.

**Example 2.2.4.** *The parameters on the left side of the ill-typed function* `negateIf` *are in the wrong order.*

```
negateIf :: Num a ⇒ Bool → a → a
negateIf x b = if b then x else negate x
```

*The following constraints will eventually be passed to the solver.*

1. *Given class constraint (type signature):* `Num a`
2. *Wanted class constraint (using* `negate` *on* `Bool`*):* `Num Bool`
3. *Wanted equality constraint (returning type* `Bool` *instead of* `a`*):* `a ~# Bool`

Note that the type variable `a` in the third constraint is fixed by the type signature of `negateIf` and cannot be unified with `Bool`. While it is also easy for a human to see that there is no way to solve the equality constraint, the type checker does not know how the type constructor `Bool` is defined. If `Bool` were a type family, there might be an instance that makes the constraint solvable, which is why the equality gets sent to the constraint solver for further inspection. To make the programming of complex plugins to GHC's type system easier, GHC will always create an equality constraint for a failed unification, even if the compiler can detect that there is no way to solve the constraint. In such cases, the constraint will contain a marker to prevent the solver from wasting time to disprove the equality.

For the example above, the constraint solver will notice that there is no rule that makes `Bool` equal to `a` and no instance `Num Bool`. The first unsolved constraint will result in the error message that GHC could not match expected type `Bool` with actual type `a`. The error message from the second constraint will not be reported to the user, because GHC correctly thinks that a more important error has occurred for the same set of constraints. By turning on the debugging flag `-ddump-tc-trace`, we can still see that the constraint was reported as incorrect by the constraint solver.

# 3. Design

The *Curry-Plugin* compiles selected modules using a nondeterministic effect with call-time choice semantics. In this chapter, we will illustrate how to use the plugin and explain some of the design choices we made during development.

## 3.1. Overview

Figure 3.1 shows an adapted version of the example program from Section 2.1 to nondeterministically compute any permutation of a given input list. Based on this example we explain the basic usage of the *Curry-Plugin*. The usage of the GHC options pragma as seen in the first line of the example activates the plugin for the given module only. In order to avoid that deterministic functions are used accidentally, we disable the implicit import of Haskell's Prelude via a language pragma. Although our newest version of the plugin automatically enables the required language pragma `NoImplicitPrelude`, we still consider it a good practice to explicitly use the language extension in the source code. While those two pragmas enable the nondeterministic transformation, we still need to bring the choice operator into scope, along with some other primitive definitions. These additional definitions are collected in a single module (`Plugin.CurryPlugin.Prelude`) that acts as a replacement for Haskell's default Prelude.

Instead of activating the plugin for the whole module, we also thought about lifting only functions that were marked with a special annotation by the user. While this design would allow for a tighter integration between deterministic and nondeterministic code, checking if a function definition only mentions compatible functions would have been more involved. Having a module with mixed definitions would also prevent us from using any kind of constraint solver plugin for the nondeterministic compilation, because there is no *reliable* way to detect, which constraint originated from a (non-)deterministic definition.

### 3.1.1. Pattern Match Semantics

Other compilers for Curry allow their users to implement the `insert` function as seen in Figure 3.1 by using overlapping patterns instead of the choice operator. However, a one-to-one copy of the overlapping implementation of `insert` would not behave as intended using *Curry-Plugin*, because of a semantic difference for overlapping pattern between the plugin and other Curry compilers. For our *Curry-Plugin*, we decided to mostly stick with the pattern match semantics from Haskell for two reasons:

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

import Plugin.CurryPlugin.Prelude

permutations :: [a] → [a]
permutations []     = []
permutations (x:xs) = insert x (permutations xs)
  where insert e []     = [e]
        insert e (x:xs) = (e:x:xs) ? (x : insert e xs)

examplePermN :: Int → [Int]
examplePermN n | n ≥ 0 = permutations [1..n]
```

**Figure 3.1.:** Nondeterministic computation of all permutations for a given list

- Curry's pattern match semantics are specific to a nondeterministic language. Implementing them would prevent us from generalizing the plugin to work with more effects than just nondeterminism.
- Using wildcards as a catch-all pattern is often used by Haskell programmers. Curry's overlapping pattern semantics would force those developers to adapt a different coding style.

There is only one small difference between Haskell and our plugin. While Haskell matches on patterns in a left-to-right fashion, the *Curry-Plugin* first looks for a position where each rule contains a constructor pattern. These positions are called *inductive* [Hanus (Ed.), 2016] and their parameters are guaranteed to be evaluated to determine the correct function rule for further evaluation. If no inductive position is found, the plugin falls back to a left-to-right pattern matching. Matching on inductive positions first reduces the risk of unnecessarily forcing the evaluation of a ⊥ value or an infinite loop as shown in the following example.

**Example 3.1.1.** *The following definition* `test` *does not terminate for left-to-right pattern matching, but it produces a result when matching on the inductive pattern of the second argument in* `(&&)` *first.*

```
test :: Bool
test = loop && False
  where loop = loop

(&&) :: Bool → Bool → Bool
True  && True  = True
_     && False = False
False && True  = False
```

```
{-# LANGUAGE TemplateHaskell #-}
module HaskellWrap where

import Example
import Plugin.CurryPlugin.Encapsulation

main = putStrLn $
    "There are " ++ show (length resTH) ++
    " permutations of a list of length " ++ show n ++
    ". Permuting an empty list results in: " ++ show resFix
  where
    n      = 9
    resTH  = $(evalGeneric DFS 'examplePermN) n -- TemplateHaskell
    resFix = eval1 DFS permutations [] :: [[Int]]
```

**Figure 3.2.:** Capturing a nondeterministic computation

Note that matching on inductive positions first might change the order of evaluation. In the context of order-dependent side effects like tracing, the changes in our matching strategy might lead to differences in the order that those effects are executed. However, in this thesis we focus on nondeterminism, where the order of effects is not observable by the programmer.

### 3.1.2. Encapsulating Nondeterministic Computations

While Haskell modules cannot be imported into Curry modules, the other direction is possible. In fact, we consider it to be best practice to only use the plugin for those parts of a program that are supposed to be nondeterministic. Any further computations and IO can be handled in a Haskell module.

To allow users of our plugin to handle the results of a nondeterministic computations in an ordinary Haskell module, we have to provide an interface to encapsulate a nondeterministic function. Figure 3.2 shows two different methods of encapsulating a computation: One that uses Template Haskell to generate a wrapper function of the correct arity, and one that uses a pre-built wrapper for a fixed arity. Especially for inexperienced Haskell programmers, the more simple functions that do not use meta-programming with Template Haskell seem to be more intuitive. This is why the plugin provides the fixed-arity wrappers for an arity of up to three.

All of the encapsulation functions take a search strategy as their first parameter and return a list of the collected results. Currently, the "Encapsulation" module provides depth-first and breadth-first search strategies, while the addition of more involved strategies is possible as well.

Each of the wrappers provided by the plugin has restrictions concerning the kind of functions that it can be used with. The implementation of `evalGeneric` uses the reification mechanism provided by Template Haskell, which dictates that the type of

the wrapped function has to be known prior to type checking the wrapper expression itself. In general, obtaining the type will never fail if the nondeterministic function is imported[1]. In addition to this Template Haskell restriction, each wrapped function is not allowed to have a higher-order function as a parameter. Higher-order functions like `map` are not excluded from being encapsulated by that restriction, because the function passed to `map` does not have a higher-order type. A few examples for restricted functions are given below in Example 3.1.2.

**Example 3.1.2.** *Although the type of the encapsulated function* `f` *in the first example is given in the type signature, Haskell does not know if that type is correct. Thus, this example will not be accepted.*

```
myEvalWrong :: Nondet (Int :→ Int) -- ^ Nondeterministic function
            → Int → [Int]
myEvalWrong f n = $(evalGeneric DFS 'f) n
```

*When a specialized wrapper without TH is used instead, the example is accepted.*

```
myEvalOkay :: Nondet (Int :→ Int) -- ^ Nondeterministic function
            → Int → [Int]
myEvalOkay f n = eval1 DFS f n
```

*The next function is accepted, because* ***map*** *has no higher-order parameter, …*

```
evalHOOkay :: [Int]
evalHOOkay = $(evalGeneric DFS 'map) (+1) [1..3]
```

*… but wrapping this fictional function is rejected, because it essentially expects a higher-order function like map as its parameter.*

```
evalHOWrong :: [Int]
evalHOWrong = $(evalGeneric DFS 'someFictionalFunc) map [1..3]
-- someFictionalFunc :: Nondet (((a :→ b) :→ [a] :→ [b]) :→ [a] :→ [b])
```

### 3.1.3. Other Features and Restrictions

In contrast to existing Curry compilers, our plugin does not support free variables, because we cannot change the syntax which is accepted by GHC. As an alternative, we could implement the `Data` type class from Hanus and Teegen [2020] that we mentioned in Section 2.1, but this has not been done yet.

To make sure that only plugin-compiled functions can be used in a Curry module, our plugin marks each compiled module with an annotation and checks if each imported module has the same annotation as well. For some modules that contain internal definitions, we set the annotation by hand.

Our *Curry-Plugin* also allows the definition of type synonyms, data types, newtypes and type classes as shown in Figure 3.3. Any data type can also use a deriving clause to create class instances automatically, but this is restricted to mostly the same classes

[1] It also works if the function is from a different *declaration group*, i.e. a group of declarations created by a top-level TemplateHaskell splice, plus all plain Haskell declarations until the next splice.

```haskell
data Queue a = Queue [a] [a]  -- ^ A data structure for representing queues.
  deriving Show

class IsList l where          -- ^ Type class for "list-like" data structures
  fromList :: [a] → l a

instance IsList Queue where   -- ^ Type class instance for 'Queues'
  fromList s = Queue s []
```

**Figure 3.3.:** Using datatypes and type classes

as in Haskell. Our current implementation is, however, not able to derive instances of `Read`. While we do not officially support any language extensions yet, multi-parameter type classes and other extensions to Haskell's type classes are deemed compatible with the plugin. A comprehensive list of unofficially supported language extensions can be found in Appendix C.

When importing a plugin-compiled module in an ordinary Haskell module, all of the defined data types are available as the normal deterministic version. This allows a user to work with the result of encapsulated functions as usual, without having to manually convert a nondeterministic data type.

While data types are available for further use, any defined or derived instances and type classes are only provided as their nondeterministic version, because converting a nondeterministic function to a deterministic one is not possible in general. This restriction also applies to datatypes that contain functions. They can not be converted automatically. To use any instances from a plugin module in the deterministic world, the user has to define them again manually or by using a standalone deriving instance. While not having a `Show` instance might be inconvenient when using GHCi to inspect the result of a function, there are no good solutions to this problem yet.

## 3.2. Using other Effects

Our plugin can also be adapted to other monadic effects by changing the code in two key modules: The effect implementation in `Plugin.Effect.Monad` has to be swapped and new encapsulation functions need to be implemented to replace the functions in `Plugin.CurryPlugin.Encapsulation`. The same lifting mechanism works for every monadic effect and can be used to implement nondeterminism, probabilism, reactive programming [Van Der Ploeg, 2013] and other effects.

As a proof-of-concept we have forked our plugin[2] and implemented a new plugin for probabilism. Its monadic effect type is implemented via algebraic effects [Plotkin and Pretnar, 2009] and a free monad construction, but that was not part of my work. The implementation of the plugin provides its user with the primitive probabilistic function `choice :: Double -> a -> a -> a`, where `choice p x y` chooses `x` with a

[2]Code available at https://git.ps.informatik.uni-kiel.de/kaiprott/2020-kprott-ma/.

probability of p and y with counter-probability $(1 - \texttt{p})$. Note that the fork is based on an older version of the plugin. The implementation is just a proof-of-concept and is not expected to be fully functioning. The function definitions in the following Example 3.2.1 allow us to compute the probability that a random string of a given length over the alphabet containing "a" and "b" is a palindrome.

**Example 3.2.1.** *Using the* `choice` *primitive, we can define a function to pick a value from a given list using a uniform probability distribution.*

```
uniform :: [a] → a
uniform [x]    = x
uniform (x:xs) = choice 0.5 x (uniform xs)
```

*With our* `uniform` *function, we can implement a function to pick a character from our alphabet "ab" and a function to create a random string with* `n` *independently chosen charactes.*

```
pickChar :: Char
pickChar = uniform ['a', 'b']

randomString :: Int → String
randomString n | n == 0 = ""
               | n >  0 = pickChar : randomString (n-1)
```

*To check if a given string is a palindrome, we compare it with its reverse.*

```
isPalindrome :: String → Bool
isPalindrome s = s == reverse s
```

*If we encapsulate a computation that checks if a random string of length two is a palindrome, we can see that the string is a palindrome with a probability of 50%.*

```
randomPalindrome :: Bool
randomPalindrome = isPalindrome (randomString 2)

-- ghci> print (allValuesNF randomPalindrome)
-- Dist [(True, 0.5), (False, 0.5)]
```

In the future, we would like to have a single plugin, which is usable for any effect without a hard-coded effect implementation. These plans for a generalized plugin will be discussed as part of the future work in Section 6.3.

# 4. Implementation

In this chapter, we will explain how the *Curry-Plugin* achieves its semantic transformation. The plugin consists of three sub-plugins, each having a different purpose and using a different extension point of GHC's plugin API as shown in Figure 4.1. The "Import Constraint Solver" and the "Import Check" are both concerned with the correct handling of imports and will be explained in Section 4.6, while the main transformation takes place in the "Lifting" phase. The lifting can be divided into five sub-phases:

1. Lifting type constructors (Section 4.1.1 and Section 4.4.1) to allow deeply nested nondeterminism in data structures.

2. Compiling and simplifying pattern matching (Section 4.2) to make the implementation of phase five simpler and more manageable.

3. Deriving internal type classes for all data types (Section 4.3), because our transformation requires each data type to have certain instances[1].

4. Lifting instance information (Section 4.4.2) of existing type class instances to account for the changed data type definitions after the first phase.

5. Lifting function definitions (Section 4.1.2) to achieve our desired semantics.

Although functions are lifted last in the plugin, knowing how they are lifted is crucial in understanding earlier phases, which is why we will start this chapter with an introduction to the lifting of types, data types and functions.

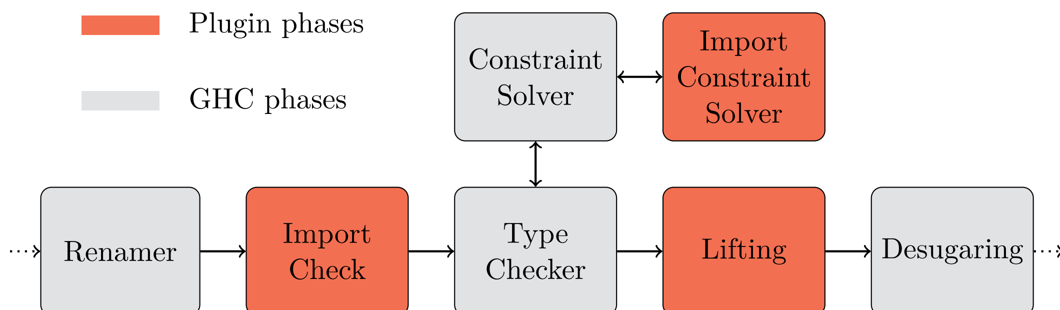


**Figure 4.1.:** Extension points used by the *Curry-Plugin*

[1]Technically, internal type classes are derived before pattern match compilation takes place.

## 4.1. Monadic Lifting

At the core of our plugin transformation stands a monadic lifting, where the monad is instantiated with our preferred effect to achieve the desired semantics. In order to lift data types and functions, we first have to know how types are lifted in general. For our *Curry-Plugin*, we are using `Nondet` as our monadic type constructor, where `Nondet` is just a newtype wrapper around the nondeterminism implementation by Fischer, Kiselyov, and Shan [2011]. We could use any other monadic effect implementation for nondeterminism with call-time choice, but settled for the mentioned library because it is relatively fast and easy-to-use. Our main lifting $[\![\cdot]\!]$ requires two other operations:

- $[\![\cdot]\!]^i$, a lifting for "inner" types, which does not wrap the full type and
- $[\![\cdot]\!]^r$, an operation that just replaces type constructors.

The main lifting works by wrapping the whole type without constraints in our monad and replacing every type constructor, if available, with its nondeterministic version. Haskell's special function type constructor `(→)` is also replaced, by applying the lifting to both sides of the the arrow. This will automatically wrap both sides with the monadic type constructor as well. For better readability, we will use the type synonym.

```
type (:→) a b = Nondet a → Nondet b
```

To avoid wrapping any constraints or variable quantifier, we first split off any invisible Π-types, where an invisible Π-type is a variable binder bound by `forall` or a (type class) constraint. Invisible means that the type or constraint does not have to be explicitly applied to this type. With `PolyKinds`, GHC allows writing types that require their type applications to be visible. Only wrapping the un-quantified part of the type prevents the rank (i.e. the nesting of a `forall` type; see [Odersky and Läufer, 1996]) from increasing, which keeps the type as simple as possible. The full lifting scheme for types is given in Figure 4.3, with examples for it in Figure 4.2.

The fact that we avoid wrapping invisible Π-types is present in the lifting rules for `forall` quantifiers and constraints, where we use a different lifting if the inner part of the type still contains constraints or quantifiers. Our lifting scheme has no rules for the type literals, kind casts or type-level injected coercions that were presented as part of GHC's internal type system in Section 2.2, because those are only used with certain language extensions. One detail to note about the inner lifting $[\![\cdot]\!]^i$ is that its rule is only applicable to monomorphic types without constraints and quantifiers. We never use the inner lifting for anything else, so this is sufficient for now.

```
-- Int has no ND version, while List and Num have one
             ⟦ [Int] ⟧ = Nondet (ListND Int)
         ⟦ a → [Int] ⟧ = Nondet (a :→ ListND Int)
   ⟦ forall a. a → a ⟧ = forall a. Nondet (a :→ a)
⟦ forall a. Num a ⇒ a ⟧ = forall a. NumND a ⇒ Nondet a
```

**Figure 4.2.:** Examples for the lifting of types

### Full Lifting

$[\![\ \texttt{forall v. ty}\ ]\!] = \texttt{forall v.}\ [\![\ \texttt{ty}\ ]\!]$ if $\texttt{ty}$ is a $\texttt{forall}$ or $(\Rightarrow)$

$[\![\ \texttt{forall v. ty}\ ]\!] = \texttt{forall v. Nondet}\ [\![\ \texttt{ty}\ ]\!]^i$ if $\texttt{ty}$ is not a $\texttt{forall}$ or $(\Rightarrow)$

$[\![\ \texttt{ty}_1 \Rightarrow \texttt{ty}_2\ ]\!] = [\![\ \texttt{ty}_1\ ]\!]^r \Rightarrow [\![\ \texttt{ty}_2\ ]\!]$ if $\texttt{ty}_2$ is a $\texttt{forall}$ or $(\Rightarrow)$

$[\![\ \texttt{ty}_1 \Rightarrow \texttt{ty}_2\ ]\!] = [\![\ \texttt{ty}_1\ ]\!]^r \Rightarrow \texttt{Nondet}\ [\![\ \texttt{ty}_2\ ]\!]^i$ if $\texttt{ty}_2$ is not a $\texttt{forall}$ or $(\Rightarrow)$

$[\![\ \texttt{ty}_1 \rightarrow \texttt{ty}_2\ ]\!] = [\![\ \texttt{ty}_1\ ]\!] \rightarrow [\![\ \texttt{ty}_2\ ]\!]$

$[\![\ \texttt{ty}_1\ \texttt{ty}_2\ ]\!] = \texttt{Nondet}\ ([\![\ \texttt{ty}_1\ ]\!]^i\ [\![\ \texttt{ty}_2\ ]\!]^i)$

$[\![\ \textit{type_constr}\ ]\!] = \texttt{Nondet}\ \textit{nd_type_constr}$

$[\![\ \textit{type_variable}\ ]\!] = \texttt{Nondet}\ \textit{type_variable}$

### Inner Lifting

$[\![\ \texttt{ty}\ ]\!]^i = \texttt{ty}'$ where $[\![\ \texttt{ty}\ ]\!] = \texttt{Nondet}\ \texttt{ty}'$

### Type Constructor Replacement

$[\![\ \texttt{forall v. ty}\ ]\!]^r = \texttt{forall v.}\ [\![\ \texttt{ty}\ ]\!]^r$

$[\![\ \texttt{ty}_1 \Rightarrow \texttt{ty}_2\ ]\!]^r = [\![\ \texttt{ty}_1\ ]\!]^r \Rightarrow [\![\ \texttt{ty}_2\ ]\!]^r$

$[\![\ \texttt{ty}_1 \rightarrow \texttt{ty}_2\ ]\!]^r = [\![\ \texttt{ty}_1\ ]\!]^r \rightarrow [\![\ \texttt{ty}_2\ ]\!]^r$

$[\![\ \texttt{ty}_1\ \texttt{ty}_2\ ]\!]^r = [\![\ \texttt{ty}_1\ ]\!]^r\ [\![\ \texttt{ty}_2\ ]\!]^r$

$[\![\ \textit{type_constr}\ ]\!]^r = \textit{nd_type_constr}$

$[\![\ \textit{type_variable}\ ]\!]^r = \textit{type_variable}$

**Figure 4.3.:** Lifting rules for types

### 4.1.1. Lifting Data Types

In order to support deeply nested nondeterminism in data types, every constructor has to be lifted. If we look at the type of those constructors, we might be tempted to apply our lifting to the full type. While this will be necessary to lift functions in Section 4.1.2, we know that the (partial or full) application of a constructor can never introduce any new effects by itself. This allows us to only lift the parameters of each constructor, because they are the only potential sources of nondeterminism in a data type. In fact, applying the lifting to the full type of a constructor would result in invalid data type definitions, because a constructor has to have its corresponding type constructor as a result in Haskell. The following code shows this restriction by defining a `Maybe` data type and its incorrectly lifted version with GADT syntax.

```
data Maybe a where                  data MaybeND a where
  Nothing :: Maybe a                  NothingND :: Nondet (MaybeND a)
  Just    :: a → Maybe a              JustND    :: Nondet (a → Nondet (MaybeND a))
```

When compiling the lifted definition `MaybeND`, GHC will complain that "data constructor `NothingND` returns type `Nondet (MaybeND a)` instead of an instance of its parent type `MaybeND a`". We will later see in the lifting of function implementations that we have to treat constructors different than functions, because their types have a different structure.

According to the design choices explained in Chapter 3, we also want to keep the original data type definition. This forces us to rename our nondeterministic version by adding "ND" as a suffix to types, constructors and record fields. This is done without checking for name clashes between new and existing data types, because the compiler uses a unique key and not the name to differentiate between identifiers after GHC's renaming phase. If one of the types with a name clash is imported somewhere else, it is also clear that the user wants to import the deterministic type. In order for GHC to know the correct type as well, we decided to only export original data type definitions, because they are the ones that should be mentioned in the source code. We can now define the following rule to lift data types, an example for it is given in Figure 4.5.

$$\left[\!\left[\begin{array}{l}\texttt{data D } a_1 \ldots a_n \\ \quad = C_1 \ ty_{11} \ \ldots \ ty_{1n} \\ \quad \mid \ \vdots \\ \quad \mid C_m \ ty_{m1} \ \ldots \ ty_{mn}\end{array}\right]\!\right]^d \Rightarrow \begin{array}{l}\texttt{data DND } a_1 \ldots a_n \\ \quad = CND_1 \ [\![ ty_{11} ]\!] \ \ldots \ [\![ y_{1n} ]\!] \\ \quad \mid \ \vdots \\ \quad \mid CND_m \ [\![ ty_{m1} ]\!] \ \ldots \ [\![ y_{mn} ]\!]\end{array}$$

**Figure 4.4.:** Transformation rule for data type definitions

**Type synonyms** As type synonyms are just aliases for a concrete type, it seems reasonable to apply the normal lifting for types to the right side of each synonym definition. While this could certainly work, it introduces a few problems. This can be seen in the following example, where we define and use a `String` type synonym on the left, with the lifted versions on the right.

```
-- Unlifted                               -- Lifted
data Maybe a                              data MaybeND a
  = Nothing                                 = NothingND
  | Just a                                  | JustND (Nondet a)

-- In "pseudo" Haskell                    -- Provided as a built-in definition
data [] a                                 data ListND a
  = []                                      = NilND
  | a : [a]                                 | ConsND (Nondet a) (Nondet (ListND a))

-- Unlifted, in GADT syntax               -- Lifted, in GADT syntax
data Either l r where                     data EitherND l r where
  Left  :: l → Either l r                   LeftND  :: Nondet l → EitherND l r
  Right :: r → Either l r                   RightND :: Nondet r → EitherND l r
```

**Figure 4.5.:** Examples for the lifting of `data` types

**Example 4.1.1.** *The following lifting for type synonyms is not correct.*

```
-- Unlifted                               -- Lifted
type String = [Char]                      type StringND = Nondet (ListND Char)
isEmptyStr :: String → Bool               isEmptyStr :: Nondet (StringND :→ Bool)
```

*If we now expand both the synonyms* `StringND` *and* `(:→)` *in the nondeterministic type signature of* `isEmptyStr`*, we end up with the following type.*

```
Nondet (Nondet (Nondet (ListND Char)) → Nondet Bool)
```

*The expanded type signature contains one more application of* `Nondet` *to the first parameter* `ListND Char` *than we would like to have, because both the lifting of the type signature and the lifting of* `String` *introduced a* `Nondet` *application.*

By lifting type synonyms with the inner lifting defined in Figure 4.3 instead, we prevent this duplicate inclusion of `Nondet`.

**Newtypes** The constructor of a newtype declaration has different semantics when compared to a constructor from a `data` declaration: At run time, a newtype constructor is desugared to a coercion between the types on the left and right side of its declaration. This makes the behavior of a newtype more similar to a type synonym than a `data` declaration. The notable difference is, that type class instances cannot be defined for type synonyms, only for data and newtypes. Originally, our lifting treated newtype declarations as ordinary data declaration, which turned out to be wrong. To see this, let us take a look at a newtype-version of the example from above.

```
-- Unlifted                               -- Lifted
newtype String = Str [Char]               newtype StringND = Str (Nondet (ListND Char))
isEmptyStr :: String → Bool               isEmptyStr :: Nondet (StringND :→ Bool)
```

```
-- Unlifted                              -- Lifted
newtype Ident   a = Ident a              newtype IdentND  a = IdentND a
newtype IDFun   a = IDFun (a → a)        newtype IDFunND  a = IDFunND (a :→ a)
type Invisible a = a                     type InvisibleND a = a
type Arrow    a b = a → b                type ArrowND    a b = a :→ b
```

**Figure 4.6.:** Examples for the lifting of `newtypes` and type synonyms

We have already established, that a newtype behaves just like a type coercion at run time. This also implies, that a type constructor from a newtype declaration is replaced with its right side during desugaring and we end up with the same problematic type as in the synonym example. By using the inner lifting for newtypes instead of the lifting for data types, this problem can be avoided[2] In Figure 4.6 we give examples for the correct lifting of a few type synonyms and newtype declarations.

**Replacing Type Constructors** When lifting any type throughout the plugin, we can use a map to lookup a nondeterministic replacement for an occurrence of a type constructor, except in the lifting of type constructors itself. In GHC's datatype for type constructors, the entities (e.g., data types, classes, …) behind each definition refer back to their "parent" type constructor. This leads to a cyclic, graph-like data structure that is convenient to use, but difficult to update. If we lift the definition of a data type or any other type constructor, we need to perform this replacement without having the final type constructor at hand; the input of our lifting depends on its output. This also happens if we look at only a single data type definition: The result type of each constructor definition is exactly the type constructor being defined. This is shown more clearly in the following example, by using GADT syntax for the type definition.

```
data Identity a where
  Identity :: a → Identity a
                 -- ^ The value constructor inside the data declaration
                 --   refers back to the type constructor being defined.
```

Thankfully, we can use Haskell's lazy evaluation to make the lifting of type constructors a simple fixed-point computation by feeding the output of the function back to its input. We only have to be careful with printing debug information to avoid forcing any results. Our data type lifting also requires some information behind IO-accessible references, so we also have to be careful with the order of some operations to avoid deadlocks.

### 4.1.2. Lifting Functions

If we take a look at the type of a lifted function, we can already see how the lifted implementation of that function has to look like.

[2] Section 4.4 will introduce a small exception to the treatment of newtype definitions that are introduced by the dictionary transformation of type classes.

1. Every lambda abstraction has to be wrapped in a `return` statement, because each type-level arrow is nested inside a `Nondet`. Additionally, each lambda binder has to be unary and no variables can be bound on the left side of a function definition, otherwise we have no place to add the `return`.

2. Before pattern matching on a value, we have to first extract it from the monad using bind (>>=). This also implies, that pattern matching in a lambda or nested pattern matching is not possible, because the nested values are nondeterministic as well (except for newtypes).

3. Before applying a function to a value, we first have to extract the function from the monadic wrapper using bind again. This has to be done for each parameter that is passed to it.

While the last point can easily be achieved during lifting, the first two are hard to implement without some kind of pre-processing. This is why we assume in the lifting of functions, that every input function only contains simple, non-nested pattern matching without lazy patterns, unary lambda expressions with only variable patterns and no variables on the left side of its definition. For pattern matching, we even go a step further and assume that the outermost pattern is only a constructor and never a variable. Example 4.1.2 shows a few examples for what is and is not expected to occur in the input of our lifting. The pre-processing itself will be explained in Section 4.2.

**Example 4.1.2.** *The following functions show the (un-) expected input for the lifting of functions. Let us consider the three given implementations of* `const`*. Only the third implementation is allowed in the lifting, because the first one contains pattern on the left side of a function definition while the second contains a non-unary lambda.*

```
const1 x y = x           -- ^ Not allowed
const2 = λx y → x        -- ^ Not allowed
const3 = λx → λy → x     -- ^ Allowed
```

*Any matching on constructor pattern is only allowed to occur in case expressions. Thus, only the second variant of* `isJust` *is valid after pre-processing.*

```
-- | Not allowed
isJust1 (Just _) = True
isJust1 _        = True
-- | Allowed
isJust2 = λx → case x of
  Just _ → True
  _      → False
```

*Nested constructor patterns as used below in* `unwrapSingleton` *are forbidden, multiple case expression should be used instead.*

```
-- | Not allowed
unwrapSingleton1 = λ(x : []) → x
-- | Allowed
unwrapSingleton2 = λx → case x of { (x : xs) → case xs of { [] → x } }
```

In the following paragraphs, we will explain how the lifting of functions, variables, lambda abstraction, application, constructors, case expressions and let expressions works. Every other syntactic element, like if-expressions or operator applications, can be translated by adapting the same techniques, which is why we skip their translation. We also give the translated type annotations and applications for every syntactic element that is annotated in GHC. These type annotations are not from a user-written type annotation, as any user-writtern type is irrelevant after the type check (see Section 2.2). For better readability, we omit the Monad dictionaries that would be passed to `return/(>>=)` and sometimes omit their type annotations if they are clear from the context. A full example for the translation can be found in Appendix A.

**Variables** By assuming that every ocurrence of a variable outside of patterns is lifted, mostly the type of a variable has to be edited. In contrast to data types, we cannot keep the original version of lifted functions, because we do not know if the original definitions used any nondeterministic function or primitive. So a renaming of functions and variables is not reqired. Later we will see in Section 4.3.3, that lifted variables need to be supplied with new or updated dictionaries. Additionally, a few of GHC's built-in functions will be replaced with lifted versions as discussed in Section 4.5. Both of these extensions are not part of the core lifting and will be discussed later.

$$[\![\ \texttt{v :: ty}\ ]\!]^e = \texttt{v :: } [\![\ \texttt{ty}\ ]\!]$$

**Figure 4.7.:** Transformation rule for variables

**Example 1.** *The variable* `isOdd` *is lifted as shown below.*

$[\![\ \texttt{isOdd :: Int} \rightarrow \texttt{Bool}\ ]\!]^e = \texttt{isOdd :: Nondet (Int :}\!\rightarrow\texttt{ Bool)}$

**Lambda Abstractions** A lambda abstraction is translated by simply wrapping it in a correctly-typed `return` and translating the inner expression. If the variable bound by the lambda occurs more than once on the right side, we also insert a `share` operator to explicitly share that variable across all its occurrences. This operator and the reason why we need it will be explained in Section 4.3.

$$\begin{aligned}
&[\![\ (\lambda(\texttt{x :: ty}_1) \rightarrow \texttt{e :: ty}_2)\ ]\!]^e \\
&\quad = \texttt{return @ Nondet @ } [\![\ \texttt{ty}_1 \rightarrow \texttt{ty}_2\ ]\!]^i \\
&\qquad\quad (\lambda(\texttt{x' :: } [\![\ \texttt{ty}_1\ ]\!]) \rightarrow \texttt{share x'} \gg\!= \lambda\texttt{x} \rightarrow [\![\ \texttt{e}\ ]\!]^e \texttt{ :: } [\![\ \texttt{ty}_2\ ]\!]
\end{aligned}$$

**Figure 4.8.:** Transformation rule for lambda abstractions

**Example 2.** *A lambda expression that implements the identity function is lifted as shown below.*

$[\![\ \lambda\texttt{x} \rightarrow \texttt{x}\ ]\!]^e = \texttt{return } (\lambda\texttt{x'} \rightarrow \texttt{share x'} \gg\!= \lambda\texttt{x} \rightarrow \texttt{x})$

**Applications** The transformation of an application of two expressions is straightforward, we only have to extract the "real" function from the monad before applying it in the lifted setting.

```
⟦ (e₁ :: ty₁ → ty₂) (e₂ :: ty₁) ⟧ᵉ
  = (>>=) @ Nondet @ ⟦ ty₁ → ty₂ ⟧ⁱ @ ⟦ ty₁ ⟧ⁱ)
      ⟦ e₁ ⟧ᵉ (λ(f :: ⟦ ty₁ → ty₂ ⟧ⁱ) → f ⟦ e₂ ⟧ᵉ :: ⟦ ty₂ ⟧)
```

**Figure 4.9.:** Transformation rule for applications

**Example 3.** *The application of* `isOdd` *onto the variable* `zero` *is lifted as shown below.*

```
⟦ isOdd zero ⟧ᵉ = isOdd >>= λf → f zero
```

**Constructors** While the type of a lifted constructor for a type $\texttt{T}\ a_1 \ldots a_n$ has the form $\texttt{Nondet}\ \tau_1 \rightarrow \ldots \rightarrow \texttt{T}\ a_1 \ldots a_n$, our functions are expected to have a type of $\texttt{Nondet}\ (\texttt{Nondet}\ \tau_1 \rightarrow (\ldots \rightarrow \texttt{Nondet}\ (\texttt{T}\ a_1 \ldots a_n)))$ This type discrepancy can be solved in two ways: We could make our translation rule for applications more complex by adapting it to work for fully saturated constructor applications as well. Any partial constructor applications would be transformed into anonymous lambda functions. While this would generate a concise and simple lifting output, it is hard to implement. Instead, we decided to generate more complex code and hope for GHC to optimize it sufficiently. Our implementation transforms ocurrences of a constructor to have a similar type to a function. To achieve this, we create a chain of `return` applications and lambda binders, applying the lambda-bound variables to the nondeterministic constructor in the end. For newtypes, however, we have to keep in mind that the value inside a newtype constructor is unlifted. Thus, the plugin cannot apply the nondeterministic newtype constructor to the lifted variable. Instead, our *Curry-Plugin* generates a call to `fmap` to apply the constructor onto its single argument inside the monadic context.

**Data Constructors**

```
⟦ C :: ty₁ → ... → tyₙ ⟧ᵉ
  = return @ Nondet @ ⟦ ty₁ → ... → tyₙ ⟧ⁱ $ λ(x₁ :: ⟦ ty₁ ⟧) →
    ...
    return @ Nondet @ ⟦ tyₙ ⟧ⁱ $ CND x₁ ... xₙ₋₁
```

**Newtype Constructors**

```
⟦ C :: ty₁ → ty₂ ⟧ᵉ
  = return @ Nondet @ ⟦ ty₁ → ty₂ ⟧ⁱ $ λ(x :: ⟦ty₁⟧) →
    fmap @ Nondet @ ⟦ ty₁ ⟧ @ ⟦ ty₂ ⟧ CND x
```

**Literals**

```
⟦ lit :: ty ⟧ᵉ = return @ Nondet @ ⟦ ty ⟧ⁱ lit -- Like a nullary constructor
```

**Figure 4.10.:** Transformation rules for data- and newtype constructors

**Example 4.** *The data constructor* `Just` *is lifted as shown below.*

```
⟦ Just ⟧^e = return (λx → return (JustND x))
```

**Example 5.** *The newtype constructor* `Identity` *is lifted as shown below.*

```
⟦ Identity ⟧^e = return (λx → fmap IdentityND x)
```

**Case Expressions** Although a case expression is assumed to be in a simplified form already, we still need to take care of a few things. Before performing a case analysis in the lifted setting, we need to use (≫=) to extract our value from the monad again. The translation of a case-branch is more complex: We know that the outermost pattern is definitely a constructor pattern that only contains variable patterns. In most cases these variables will be lifted, except if the constructor stems from a newtype declaration. In order to assume that every variable we encounter somewhere else in the AST is lifted, we need to introduce a lifted version for any variable in a case expression over a newtype. A lifted variable might need to be shared explicitly again, but sharing any of these unwrapped newtype variables is not required, because they can only contain an effect in a more deeply nested position. This nested effect will be shared by a case expression that scrutinizes the effectful part of this value if necessary.

**Case Expressions**

```
⟦ case (e :: ty1) of { br₁; ...; brₙ } :: ty₂ ⟧^e
  = (≫=) @ Nondet @ ⟦ ty₁ ⟧^i @ ⟦ ty₂ ⟧^i)
      ⟦ e ⟧^e (\case { ⟦ br₁ ⟧^b; ...; ⟦ brₙ ⟧^b })
```

**Case Branches - Data Constructor**

```
⟦ C (n₁ :: ty₁) ... (nₙ :: tyₙ) → e ⟧^b
  = CND (n₁' :: ⟦ ty₁ ⟧) ... (nₙ' :: ⟦ tyₙ ⟧)
  → share n₁' ≫= λn₁ →
     ...
     share nₙ' ≫= λnₙ →
     ⟦ e ⟧^e
```

**Case Branches - Newtype Constructor**

```
⟦ C (n :: ty) → e ⟧^b
  = CND (n' :: ⟦ ty ⟧^i))
  → let n = return n' in ⟦ e ⟧^e
```

**Figure 4.11.:** Transformation rules for case expressions and branches

**Example 6.** *A case expression with a data constructor is transformed as shown below.*

```
⟦ case x of { Just y → y + y; Nothing → zero } ⟧^e
  = x ≫= \case
      JustND y' → share y' ≫= λy → ⟦ y + y ⟧^e
      NothingND → zero
```

**Example 7.** *A case expression with a newtype constructor is transformed as shown below.*

```
⟦ case x of { Identity x → x } ⟧^e
  = x >>= \case
      Identity x' → let x = return x' in x'
```

**Let Expressions** Let expressions are expected to only contain functions and no pattern bindings, because the latter have been desugared during the pre-processing (see Section 4.2.3). We obviously have to translate local definitions like any other function, but we also need to share any newly introduced variables.

```
⟦ let v = e₁ in e₂ ⟧^e = let v' = ⟦ e₁ ⟧^e in share v' >>= λv → ⟦ e ⟧^e
```

**Figure 4.12.:** Transformation rule for let expressions

**Example 8.** *A let expression that binds a single variable is transformed as shown below.*

```
⟦ let x = coin in x + x ⟧^e
  = let x' = coin in share x' >>= λx → ⟦ x + x ⟧^e
```

Note that we try to split let expressions with more that one binding into nested let expression first, to make sharing across multiple let-bound definitions easier. This splitting is not possible if the expressions are (mutually) recursive, but the general lifting will still work. In Section 4.3.4 we will see, that sharing of variables across unsplittable or recursive bindings is challenging and not supported in the current plugin.

**Example 9.** *A let expression that binds two variables is transformed as shown below.*

```
⟦ let { x = zero, y = x } in x + y ⟧^e
  = ⟦ let x = zero in let y = x in x + y ⟧^e
  = let x' = zero in share x' >>= λx → let y = x in ⟦ x + y ⟧^e
```

**Optimizing Sharing for Single-Ocurrence Variables** The transformation rules given above always insert a call to `share` for bound variables to correctly model call-time choice, even if the variable does not need to be shared. To avoid sharing of single-use variables, our plugin uses different transformation rules in some cases. These optimized rules skip the insertion of a `share`, whenever the variable occurs at most once in the scope that it is bound. However, if a variable is used in multiple branches of a case-expression, our simple check does not detect whether sharing can be omitted or not.

**Example 4.1.3.** *While transforming the expression given below, our plugin detects that* `x` *needs no sharing, but fails to detect the same for* `y`*.*

```
⟦ let { x = True, y = 2 } in case x of { True → y; False → 2 * y} ⟧^e
  = let { x = True, y' = 2 } in share y' >>=
    λy → ⟦ case x of { True → y; False → 2 * y} ⟧^e
```

## 4.2. Pattern Matching

While our lifing of functions only excepts a small subset what is possible to write in Haskell, we still want to use our plugin for more complex function as well. To do this, we use a modified version of the algorithm to transform pattern matching from Wadler [1987].

That algorithm can be used to compile pattern matching to get rid of nested pattern in function rules and case expressions, but Haskell also allows patterns in guards, on the left side of `let`-bound definitions and in `do`-notation. We start by explaining the general translation of pattern matching and then continue with the details of all the other pattern constructs.

### 4.2.1. Translating Pattern Matching

The internal representation of rules in GHC uses the same data type for rules in both case expressions and functions, as well as for lambda (-case) expressions. This makes compilation of pattern matching siginificantly easier.

**Fresh variables** As a first step, we replace each top-level pattern of each rule with a fresh variable, because our main algorithm expects a list of variables to scrutinize and the list of rules with the corresponding patterns for each variable. For function definitions, these new variables are introduced by a lambda on the right side of a rule, while we use a let-definition to bind the scrutinee of a case expression to the variable[3]. The introduction of new variables also splits multi-arity lamdas and functions into unary lambda expressions. The following rules describe this transformation more formally. The bracket notation $[\![ match \ldots ]\!]^p$ is used for the pattern match algorithm and will be explained afterwards. `E` is used as a placeholder to be inserted if none of the patterns is applicable. For the case of our *Curry-Plugin*, we instantiate `E` with the nondeterministic failure (`failed`).

$$\begin{array}{l} \texttt{f}\ p_{11}\ \ldots\ p_{1n} = e_1 \\ \vdots \\ \texttt{f}\ p_{m1}\ \ldots\ p_{mn} = e_m \end{array} \quad\Rightarrow\quad \begin{array}{l} \texttt{f} = \lambda x_1 \to \ldots\ \lambda x_n \to [\![\ \texttt{match}\ [x_1,\ \ldots,\ x_n] \\ \qquad [([p_{11},\ \ldots,\ p_{1n}],\ e_1),\ \ldots \\ \qquad ([p_{m1},\ \ldots,\ p_{mn}],\ e_m)]\ \texttt{E}\ ]\!]^p \end{array}$$

$$\begin{array}{l} \textbf{case}\ v\ \textbf{of} \\ \quad p_1 \to e_1 \\ \quad \vdots \\ \quad p_m \to e_m \end{array} \quad\Rightarrow\quad [\![\ \texttt{match}\ [v]\ [([p_1],\ e_1),\ \ldots,\ ([p_m],\ e_m)]\ \texttt{E}\ ]\!]^p$$

$$\begin{array}{l} \textbf{case}\ e\ \textbf{of} \\ \quad p_1 \to e_1 \\ \quad \vdots \\ \quad p_m \to e_m \end{array} \quad\Rightarrow\quad \begin{array}{l} \textbf{let}\ v = e \\ \textbf{in}\ [\![\ \texttt{match}\ [v]\ [([p_1],\ e_1),\ \ldots,\ ([p_m],\ e_m)]\ \texttt{E}\ ]\!]^p \end{array}$$

[3]While the introduction of a new `let` would increase the number of heap allocations during execution, GHC's optimizer will inline these bindings under most circumstances.

After introducing new variables for each argument, we use the $[\![$`match xs eqs E`$]\!]^p$ transformation to desugar any pattern matching. Here, `xs` is the list of all newly introduced variables, `eqs` are the matching rules as a list of pattern/expression pairs an `E` is the expression to use if no match was successful.

**Inductive Position** Haskell would now match on the variables in a left-to-right order, but for our plugin we decided to first look for an inductive position that will defnitely be demanded as outlined in Chapter 3. To find this position, we traverse left-to-right to find the first argument, where each equation in `eqs` consists of a constructor pattern at top-level. Let $i \in [0 \mathrel{..} \texttt{length xs}]$ be the smallest index such that for all $(\texttt{ps}, \texttt{e}) \in \texttt{eqs}$ holds that $\texttt{ps}_i$ is a constructor pattern. If none such index exists, fall back to Haskell's pattern match semantics and use index $i = 0$.

**The `match` Transformation** There are four different cases to consider for the patterns at index $i$ in each of the equations. In the case of as-pattern (`@`), lazy-pattern (`~`) or bang-pattern (`!`)[4], we use the inner pattern for our decision instead. The following are modified rules from the description of Haskell's pattern matching by Hanus [2019].

1. **No matches left to consider because `xs` is empty:**
   There are no more matches to perform. We now need to check if there exists a rule for the matches made previously.
   a) **At least one equation left in `eqs`:**
      Use the first of the remaining equations to create the result expression.

      $[\![$ `match [] [(ps, e), ...] E` $]\!]^p$
      $\Rightarrow$ `e`

   b) **No equations left because `eqs` is empty:**
      It seem that none of the previous patterns could be matched successfully, so we use `E` for a catch-all alternative.

      $[\![$ `match [] [] E` $]\!]^p$
      $\Rightarrow$ `E`

2. **All equations in `eqs` use a variable pattern at index $i$:**
   In this case, we substitute the variables in each equation.

$$
\begin{array}{l}
[\![\ \texttt{match xs [(}[\texttt{p}_{11}\texttt{, ..., v}_{1i}\texttt{, ..., p}_{1n}]\texttt{, e}_1\texttt{),} \\
\qquad\qquad\qquad \vdots \\
\qquad\qquad\quad \texttt{(}[\texttt{p}_{m1}\texttt{, ..., v}_{mi}\texttt{, ..., p}_{mn}]\texttt{, e}_m\texttt{)] E}\ ]\!]^p \\[1em]
\Rightarrow [\![\ \texttt{match }[\texttt{x}_1\texttt{, ..., x}_{(i-1)}\texttt{, x}_{(i+1)}\texttt{, ..., x}_n] \\
\qquad\qquad [\texttt{(}[\texttt{p}_{11}\texttt{, ..., v}_{1(i-1)}\texttt{, v}_{1(i+1)}\texttt{, ..., p}_{1n}]\texttt{, e}_1[\texttt{v}_{1i} \mapsto \texttt{x}_i]\texttt{),} \\
\qquad\qquad\quad \vdots \\
\qquad\qquad\quad \texttt{(}[\texttt{p}_{m1}\texttt{, ..., v}_{m(i-1)}\texttt{, v}_{m(i+1)}\texttt{, ..., p}_{mn}]\texttt{, e}_m[\texttt{v}_{mi} \mapsto \texttt{x}_i]\texttt{)] E}\ ]\!]^p
\end{array}
$$

[4] Bang-pattern are not actually supported by the plugin, but the pattern match implementation already accounts for their occurence.

3. **All equations in eqs use a constructor pattern at index *i*:**
We need to create a case expression over $x_i$ and translate the rest of our matching together with any new nested patterns. To create the patterns for our new case, we first group all eqs to ($eqs_1$ ++ …$eqs_k$), such that each $ps_i$ in every $eqs_j$ starts with the constructor $C_j$, where $C_1$ …$C_k$ are all **used** constructors of the same type as $x_i$. Let each $eqs_j$ be of the following form.

$$
\begin{aligned}
&\mathtt{eqs}_j = \\
&\quad [([\mathtt{ps}_{11},\ \ldots,\ \mathtt{ps}_{1(i-1)},\ (\mathtt{C}_j\ \mathtt{p}^1_{1j}\ \ldots\ \mathtt{p}^{n_j}_{1j}),\ \ldots,\ \mathtt{ps}_{1n}],\ \mathtt{e}_{1j}), \\
&\quad \vdots \\
&\quad ([\mathtt{ps}_{m_j1},\ \ldots,\ \mathtt{ps}_{m_j(i-1)},\ (\mathtt{C}_j\ \mathtt{p}^1_{m_jj}\ \ldots\ \mathtt{p}^{n_j}_{m_jj}),\ \ldots,\ \mathtt{ps}_{m_jn}],\ \mathtt{e}_{m_jj})]
\end{aligned}
$$

Then our transformation can proceed by creating a branch for each used constructor and including any nested patterns in the rest of the translation. Instead of generating a branch for each unused constructor, we introduce a catch-all case to reduce code duplication.

$$
\begin{aligned}
&[\![\ \mathtt{match\ xs\ (eqs}_1\ +\!\!+\ \ldots\ +\!\!+\ \mathtt{eqs}_k\mathtt{)\ E}\ ]\!]^p \\
&\Rightarrow \mathtt{case}\ \mathtt{x}_i\ \mathtt{of} \\
&\qquad \mathtt{C}_1\ \mathtt{v}_{11}\ \ldots\ \mathtt{v}_{1n_1} \rightarrow [\![\ \mathtt{match\ xs'}_1\ \mathtt{eqs'}_1\ \mathtt{E}\ ]\!]^p \\
&\qquad \vdots \\
&\qquad \mathtt{C}_k\ \mathtt{v}_{k1}\ \ldots\ \mathtt{v}_{kn_k} \rightarrow [\![\ \mathtt{match\ xs'}_k\ \mathtt{eqs'}_k\ \mathtt{E}\ ]\!]^p \\
&\qquad _ \qquad\qquad\quad\ \rightarrow [\![\ \mathtt{match\ []\ []\ E}\ ]\!]^p
\end{aligned}
$$

where for all $a, b$ the variables $v_{ab}$ are fresh and $xs'_j$, $eqs'_j$ are defined as:

$$
\begin{aligned}
&\mathtt{xs'}_j\ =\ [\mathtt{v}_{j1},\ \ldots,\ \mathtt{v}_{jn_j},\ \mathtt{x}_1,\ \ldots,\ \mathtt{x}_{(i-1)},\ \mathtt{x}_{(i+1)},\ \ldots,\ \mathtt{x}_n] \\
&\mathtt{eqs'}_j = \\
&\quad [([\mathtt{p}^1_{1j},\ \ldots,\ \mathtt{p}^{n_j}_{1j},\ \mathtt{ps}_{11},\ \ldots,\ \mathtt{ps}_{1(i-1)},\ \mathtt{ps}_{1(i+1)},\ \ldots,\ \mathtt{ps}_{1n}],\ \mathtt{e}_{1j}), \\
&\quad \vdots \\
&\quad ([\mathtt{p}^1_{m_jj},\ \ldots,\ \mathtt{p}^{n_j}_{m_jj},\ \mathtt{ps}_{m_j1},\ \ldots,\ \mathtt{ps}_{m_j(i-1)},\ \mathtt{ps}_{m_j(i+1)},\ \ldots,\ \mathtt{ps}_{m_jn}],\ \mathtt{e}_{m_jj})]
\end{aligned}
$$

This gets slightly more complicated in the presence of lazy pattern, literal pattern or overloaded lists, which we omitted here because the main transformation stays the same.

4. **Constructor and variable patterns in eqs are mixed:**
This last case is translated by grouping constructor and variable patterns. Every group is then translated seperately, where subsequent groups are used to create the failure expression of its predecessor.

$$
\begin{aligned}
&[\![\ \mathtt{match\ xs\ eqs\ E}\ ]\!]^p \\
&\quad \Rightarrow [\![\ \mathtt{match\ xs\ eqs}_1\ [\![\ \mathtt{match\ xs\ eqs}_2\ [\![\ \ldots\ [\![\ \mathtt{match\ xs\ eqs}_l\ \mathtt{E}\ ]\!]^p \ldots ]\!]^p ]\!]^p
\end{aligned}
$$

Where eqs is equal to $eqs_1$ ++ ... ++ $eqs_l$ so that each group $eqs_\kappa$ is non-empty and all groups alternate between containing only constructor or only variable patterns.

**Example 4.2.1.** *We will now show the algorithm in action on the following partial* `xor` *definition.*

```
xor :: Bool → Bool → Bool
xor x     False = x
xor False True  = True
```

*Notice that the term* `xor loop False` *will not terminate if evaluated with Haskell's left-to-right strategy. But our pattern matching will identify the second argument as inductive and scrutinizes it first. The following transformation will take place:*

```
-- Introduce variables
xor = λv1 → λv2 → ⟦ match [v1, v2]
            [([x, False], x), ([False, True], True)] failed ⟧^p

-- Match on second position - all constructors → Rule 2
xor = λv1 → λv2 → case v2 of
  False → ⟦ match [v1] [([x], x)] failed ⟧^p
  True  → ⟦ match [v1] [([True], True)] failed ⟧^p
  _     → ⟦ match [] [] failed ⟧^p

-- Branch 1: No inductive position, use 0 instead - all variables → Rule 3
-- Branch 2: No inductive position, use 0 instead - all constr.   → Rule 2
xor = λv1 → λv2 → case v2 of
  False → ⟦ match [] [([], v2)] failed ⟧^p
  True  → case v1 of
            True → ⟦ match [] [([], True)] failed ⟧^p
            _     → ⟦ match [] [] failed ⟧^p
  _     → ⟦ match [] [] failed ⟧^p

-- Branch 1 and 2: List of variables is empty, at least one eqs → Rule 1 a)
-- Branch 3 and 4: List of variables is empty, no more eqs      → Rule 1 b)
xor = λv1 → λv2 → case v2 of
  False → v2
  True  → case v1 of
            True → True
            _     → failed
  _     → failed
```

Note that the last catch-all branch in the outer case is not required. We do not perform any exhautiveness check to avoid its creation. It is instead removed in GHC's desugaring to core, even if no optimizations are performed.

### 4.2.2. Translating Guards

There are three different kinds of statements that can occur inside a guard: A Boolean expression, a let-binding or a pattern-binding. The expression associated with a guard is only executed if all boolean expressions were evaluated to true and if all matches in the pattern-binding succeeded. Any newly introduced variables are scoped left-to-

right. This makes guards equivalent to nested let, if and case expressions. To desugar a guard statement, we need two expressions: One to use if the guard succeeds and one if it fails. In general, the failure expression is passed to the guard desugaring by the pattern match compilation and contains code that tries the remaining patterns or any other guards. Multiple statements are desugared by successively translating each one. The following rules show the translation of a single statement, where `success` and `fail` are the expressions to be used in the respective cases.

```
let (...)      ⇒      let (...) in success

p ← e          ⇒      case e of { p → success; _ → fail}

e              ⇒      if e then success else fail
```

The resulting expression is fed back into the pattern match compilation, to desugar any new case- or let expressions. An example for the full translation of guards is shown below.

**Example 4.2.2.** *The function* `isJust` *and its desugared variant without guards.*

```
-- Original definition
isJust :: Maybe a → Bool
isJust x | let y = x
         , Just _ ← y = True
         | otherwise  = False
```

```
-- After pattern and guard translation
isJust :: Maybe a → Bool
isJust = λx → let y = x in case y of
  Just _ → True
  _ → if otherwise then False else fail
```

### 4.2.3. Translating Pattern Bindings

A pattern binding in Haskell is supposed to be lazy, so it is equivalent to a case expression with a lazy pattern.

**Example 4.2.3.** *The following two programs are semantically equivalent. Both of them would evaluate to* `True` *and not to* $\bot$.

```
aTrue1 :: Bool
aTrue2 = let Just _ = undefined in True

aTrue1 :: Bool
aTrue2 = case undefined of { ~(Just _) → True }
```

Unfortunately, we cannot translate pattern bindings into lazy case expressions, because the lifted version of a lazy case would be strict in its effect by using ($\gg=$). Instead we have to translate lazy patterns into pattern bindings and introduce selector functions for the latter. Given an expression `let p = e`$_1$ `in e`$_2$, we first have to generate variants of `p` for each variable in it. This can be done with the nondeterministic set of rules shown in Figure 4.13. Note that there is no rule to generate a variant of a wildcard pattern and that wildcards are only generated for constructor arguments.

All variants of patterns are generated at constructor positions or at-patterns (`@`) to get rid of any unnecessary nested patterns. Otherwise, a generated pattern variant

might force too much of the binding to be evaluated, which would not be lazy enough. We also rename every variable to a fresh name to avoid name clashes.

**Example 4.2.4.** *The pattern* `(x@(Just y), (z:_))` *has three variants:*

```
(x', _)
((Just y'), _)
(_, (z':_))
```

We can now define the translation that creates a selector function for each pattern variant. Each selector uses a lambda function to match on its corresponding part of the original pattern. Here, `v` is a fresh variable, $\mathtt{p}_j$ are the pattern variants and $\mathtt{x}_j$, $\mathtt{x}'_j$ the corresponding original and renamed variables.

```
let p = e₁ in e₂      ⇒     let v  = e₁
                                x₁ = (λp₁ → x'₁) v
                                ⋮
                                xₙ = (λpₙ → x'ₙ) v
                            in e₂
```

**Example 4.2.5.** *The expression* `(let (x@(Just y), (z:_)) = e₁ in e₂)` *will be transformed to:*

```
let v = e₁
    x = (λ(x', _) → x') v
    y = (λ((Just y'), _) → y') v
    z = (λ(_, (z':_)) → z') v
in e₂
```

Note that the lambda functions still contain pattern matches and are thus not in the form required by the lifting as depicted in Section 4.1.2. This can be fixed by repeatedly applying the pattern match algorithm from this section.

$$\frac{\mathtt{p}_i \Rightarrow^v \mathtt{p}'_i}{\mathtt{C}\ \mathtt{p}_1\ \cdots\ \mathtt{p}_n \Rightarrow^v \mathtt{C}\ _\ \cdots\ _\ \mathtt{p}'_i\ _\ \cdots\ _} \quad \forall i \in [1 \ldots n] \tag{4.1}$$

$$\frac{\mathtt{p} \Rightarrow^v \mathtt{p}'}{\mathtt{v@p} \Rightarrow^v \mathtt{p}'} \tag{4.2}$$

$$\mathtt{v@p} \Rightarrow^v \mathtt{v}' \tag{4.3}$$

$$\mathtt{v} \Rightarrow^v \mathtt{v}' \tag{4.4}$$

where $\mathtt{v}'$ is a fresh variable in each rule.

**Figure 4.13.:** Nondeterministic rules to generate pattern variants

While the transformation given above is semanitcally correct, the current implementation has a few shortcomings with respect to the explicit sharing outlined in Section 4.3. All (mutually) recursive local definitions of a single let are placed together into so called "binding groups" in GHC. Our implementation always places all generated (selector) functions for a pattern binding in the same binding group, because a pattern binding is allowed to be recursive as well. This is also done for non-recursive bindings, because it was significantly easier to implement. However, we will later see in Section 4.3.4 that our explicit sharing implementation does not share values inside of binding groups. Until the implementation is fixed, no sharing is performed across variables in a single pattern binding, which can be seen on the following example.

**Example 4.2.6.** *Although call-time choice semantics would dictate that* `test` *evaluates to 2 and 4, the choice is not shared across x and y. All (incorrect) results are: 2, 3, 3 and 4.*

```
test = let (x, y) = (let v = 1 ? 2 in (v, v))
       in x + y
-- ghci> eval DFS test
-- [2, 3, 3, 4]
```

### 4.2.4. Translating Do-Notation

There are two different reasons for desugaring do-notation:

1. The syntax for do allows arbitrary pattern on the left side of a bind statement. Just like pattern in case expressions, a bind has to be desugared because it might contain nested (constructor) patterns.

2. When GHC desugars do-notation to core, it uses the (>>=) and `return` from the unlifted monad instance. We obviously want the compiler to use the lifted versions of both functions to ensure a correct result. In the case of list comprehensions, which use the same AST data types, GHC uses a specialized desugaring that does not use the monad instance for lists at all. In most cases, the functions that GHC uses for its desugaring are given in the internal syntax tree for the do-notation or comprehension, but in some crucial cases they are sadly missing. Only when `RebindableSyntax` or `MonadComprehensions` were used, then the functions are guaranteed to be present. Thus, the second goal of desugaring do-notation is to convince GHC that `RebindableSyntax` was used[5] and annotating any missing function required to desugar the syntax.

The first point is achieved by transforming the pattern in each bind statement to a variable pattern and then matching on that variable in a case expression, similar to how (lambda) functions are desugared already. For the second goal, we only need to find the relevant monadic functions and annotate them in the correct positions

[5] How this is done relatively technical and uninteresting. It depends on the exact syntax that was used in the source code

of the AST. Instead, we could also completely desugar do-notation to functions and applications only, but if we want to consider extensions like `ApplicativeDo` in the future, this desugaring would quickly get a lot more involved. We will only outline the first part of our transformation with an example below, because the second part is an uninteresting technical detail of the implementation.

**Example 4.2.7.** *Consider the following function* `nextInt` *to return and increment a counter in a state monad.*

```
data MyState = MyState Int
             | NoState

class MonadState s m where
  get :: m s
  put :: s → m ()

nextInt :: (MonadFail m, MonadState MyState m) ⇒ m Int
nextInt = do
  MyState i ← get
  let next = i + 1
  put (MyState next)
  return i
```

*Note that our plugin uses* `MonadFailDesugaring` *by default, which is why we need a* `MonadFail` *context on the type variable* `m`*. The function* `nextInt` *will be transformed to:*

```
nextInt :: (MonadFail m, MonadState MyState m) ⇒ m Int
nextInt = do
  v ← get
  case v of
    MyState i → do
      let next = i + 1
      put (MyState next)
      return i
    _         → fail -- Unless the pattern cannot fail.
```

This transformation of bind statements makes it easier to get rid of any nested pattern inside the do-notation, because we can now apply our pattern match algorithm to desugar the remaining case expression. Of course, let statements can be treated like normal let expressions as they are just syntactic sugar.

## 4.3. Sharing Effects

Sharing of nondeterministic choices in the plugin is required to faithfully represent Curry's call-time choice semantics, as established in Section 2.1. Although Haskell implicitly shares values as well, this is not sufficient to implement a monadic lifting with call-time choice. Consider the following program with a shared choice and its simple transformation.

```
shared :: Int                    sharedND :: Nondet Int
shared = x + x                   sharedND = (+) >>= λf1 → f1 x >>= λf2 → f2 x
  where x :: Int                   where x :: Nondet Int
        x = 1 ? 2                        x = (?) >>= λg1 → g1 1 >>= λg2 → g2 2
```

Even though x is shared in both the unlifted and lifted code, in the lifted setting the value being shared is a monadic computation. In the setting of nondeterminism, the result of this monadic computation can be represented as a tree, where a node corresponds to a choice and a leaf represents a deterministic result. The only thing that is shared between both ocurrences of x is this tree, but not the choice that is made outside the definition of x.

We can fix this problem by adapting a framework for explicitly sharing nondeterministic computations by Fischer, Kiselyov, and Shan, 2011. This framework provides a type class Shareable and the operation share that we have already used in Section 4.1.2 for our lifting. If we insert this sharing operator in the function from above, we end up with the following semantically correct code.

```
sharedND :: Nondet Int
sharedND = share x >>= λx' → (+) >>= λf1 → f1 x' >>= λf2 → f2 x'
  where x :: Nondet Int
        x = (?) >>= λg1 → g1 1 >>= λg2 → g2 2
```

To use `share :: (Monad m, Shareable m a) ⇒ m a → m (m a)`, we need an instance of Shareable for the shared value. In the next Section (4.3.1) we will discuss this type class and how to derive it automatically for all user-defined data types. Additionally, Haskell's polymorphism might require us to share the value of a polymorphic type. Making this possible was one of the biggest challenges during development. Section 4.3.3 and Section 4.3.2 will discuss two different attempts at overcoming the problem.

### 4.3.1. A Type Class for Sharing

The only operation provided by the Shareable type class is used to share any choices nested somewhere in the components of a data type. The type class and a typical instance are given below.

```
class Shareable m a where
  shareArgs :: (Monad n) ⇒
    (forall b. (Shareable m b ⇒ m b → n (m b))) → a → n a

data ListND a = Nil | Cons (Nondet a) (Nondet (ListND a))

instance Shareable Nondet a ⇒ Shareable Nondet (ListND a) where
  shareArgs f Nil         = pure Nil                -- Nothing to share
  shareArgs f (Cons x xs) = Cons <$> f a <*> f xs   -- Share both x and xs
```

Because the instances of this type class are very generic, we decided to use Haskell's Generic library [Jeuring et al., 2009] to simplify the instances even more. Using DefaultSignatures, we can give a default implementation for shareArgs for every

data type that has an instance of the `Generic` type class and where its generic representation has an instance of a generic version of `Shareable` called `ShareableGen`. The instances of this new class for all of the data types used in the generic representation of types need to be defined only once. Afterwards, we can derive the Generic instances for each type by using the `DeriveGeneric` language extension [Magalhães et al., 2010]. By enabling `DeriveAnyClass`, we can also derive our `Shareable` type class. The derived instances will use the generic implementation as long as `Generic` has been derived as well. The full implementation of `Shareable` with all its generic machinery is given in Appendix B.

The interface for encapsulating nondeterminism outlined in Chapter 3 requires us to convert between lifted and unlifted data types. This is done with the following type class, which also performs a *pull-tabbing* [Alqaddoumi et al., 2010] step to "pull" all nondeterminism from the arguments to the top level of a lifted data type.

```
class Monad m ⇒ Normalform m a b | a → b, b → a where
  nf    :: m a → m b -- From effectful   to effect-free
  liftE :: m b → m a -- From effect-free to effectful
```

Like with `Shareable`, this class can be derived via a similar generic mechanism.

### 4.3.2. Inferring Shareable Constraints in a Polymorphic Context

For any given function we know the type of every value that has to be shared. This allows us to deduce that the lifted version of `toPair` from the following example will need a `Shareable` constraint on the type `a`.

```
toPair :: a → (a, a)
toPair a = (a, a)

applyAndToPair :: (a → b) → a → (b, b)
applyAndToPair f = toPair ∘ f
```

The second top-level function never uses sharing itself, but it uses the function `toPair` instantiated with the type `b`. Consequently, we have to add `Shareable Nondet b` as a constraint in the lifted type of `applyAndToPair`.

Another thing to note in this simple example is, that we could have also over-approximated the required constraints by assuming that every polymorphic type variable might need to be shared. While this initially seems promising, only type variables with the simple kind `Type`[6] can have a `Shareable` constraint. As soon as any higher-kinded type variables are used, this over-appoximation is not possible anymore because there are infinitely many types that can be constructed using this type variable. To see this, consider the types that can be created by just combining the type variable `m :: Type → Type` with a nullary type constructor like `Int`:

```
m Int, m (m Int), m (m (m Int)), ....
```

[6] We are using GHC's new notation for Kinds, where `*` is replaced with the more verbose `Type`.

It is easy to see, however, that every implementation of a function only uses sharing on a finite number of types[7].

This allowed us to infer the required constraints by traversing the dependency graph of all involved functions and using a fixed-point iteration for mutually recursive declarations, which is quite similar to how types are inferred for dependent declarations in a Hindler-Milner [Hindley, 1969; Milner, 1978] based type system. The result of this inference was used during the lifting phase to manually include the correct `Shareable` constraints in the lifted type of each function.

The inference was originally implemented for a version that did not support type classes to test this approach. We thought that it could be adapted easily to also include class and instance declarations, but that turned out to be wrong. When lifting the type of a function from a type class, no implementation is avaliable to determine the required constraints, which forced us to add the constraints for a specific instance in the instance head instead of adding them to the class function. This appoach was short-sighted, because not every value of a class function can be constrained in the instance declaration. For example, we cannot add `Show a` as a constraint to the following instance declaration to trace all mapped values.

```
instance {- Show a ⇒ -} Functor [] where
  -- fmap :: (a → b) → [a] → [b]
  fmap f xs = map (λa → trace (show a) a) xs
  --                            ^ Not possible
```

### 4.3.3. Using Quantified Constraints for Polymorphic Sharing

After eventually noticing the issue of type classes for the inference of `Shareable`, we took another look at the over-approximation of required constraints. That approach did not have the problem of having to look at the implementation of a function to figure out the required constraints and would thus be usable for type classes as well. We quickly realized that although there are infinitely many combinations of a type variable `m :: Type → Type`, we only want to have an instance `Shareable Nondet (m x)` for every `x` that satisfies `Shareable Nondet x`. Using GHC's language extension `QuantifiedConstraints`[8], a constraint of that form can be written down as shown in the following examples.

**Example 4.3.1.** *Consider the following unlifted type signatures for two functions. The first function* `id` *uses a type variable of simple kind* `Type`*, while the generalized* `and` *function uses a type variable of kind* `Type → Type`.

```
id  :: a → a
and :: Foldable t ⇒ t Bool → Bool
```

[7] This is only true if none of the functions have a higher-rank type.

[8] We are relatively sure that `QuantifiedConstraints` is safe to use for our purposes, even though the extension leads to non-confluence in GHC when combined with local type equalities (see GHC bug report #17295[9]) and it requires us to enable `UndecidableInstances`

*If we take a look at their lifted types, we can see that a quantified constraint is used for the second type.*

```
id  :: Shareable Nondet a ⇒ Nondet (a :→ a)
and :: ( Foldable t,
       , forall x. Shareable Nondet x ⇒ Shareable Nondet (t x)) -- ← Quantified
    ⇒ Nondet (t Bool :→ Bool)
```

To make the construction of a type with `Shareable` constraints easier, the new constraints are added as the last constraints in every function. Their order is defined by the order of the `forall`s that bind the corresponding type variables. If the function happens to be an instance function for a type class dictionary, we have to differentiate between the variables bound by the instance and the variables bound by the type of the class function. Here, the constraints for instance variables are inserted before the `forall` bindings of the class function type. This difference between ordinary and instance function types can only be observed in Haskell's internal type representation, not in the pretty-printed debug output[10]. The following example shows the full lifting for an instance function.

**Example 4.3.2.** *Consider the following type definition for an identity on monadic types and its corresponding functor instance.*

```
data IdentityT m a = IdentityT (m a)
instance Functor m ⇒ Functor (IdentityT m) where (...)
```

*The internal name and type of the* `fmap` *implementation generated by the instance is shown below. Notice that the type variables and constraint given by the instance head occur at the start of the type.*

```
$cfmap :: forall m. Functor m
       -- ^ Instance variables
       ⇒ forall a. forall b. (a → b) → IdentityT m a → IdentityT m a
       -- ^ Function variables
```

*The lifted type will contain* `Shareable` *constraints at two different positions.*

```
$cfmap :: forall m. FunctorND m
       ⇒ (forall x. Shareable Nondet x ⇒ Shareable Nondet (m x)
       -- ^ Instance constraints
       ⇒ forall a. forall b.
         Shareable Nondet a ⇒ Shareable Nondet b
       -- ^ Function constraints
       ⇒ Nondet ((a :→ b) :→ IdentityTND m a :→ IdentityTND m a)
       -- ^ Lifted type
```

[10]Except when full debug mode is turned on with `-dppr-debug`, but the option also makes types and other debug output hard to read.

### 4.3.4. Problems with Recursion and Explicit Sharing

Consider the following program with a cyclic definition that includes a nondeterministic choice.

```
threeOnes :: [Int]
threeOnes = take 3 ones
  where ones :: [Int]
        ones = [] ? 1 : ones
```

If we consider Curry's call-time choice semantics, we can argue that `threeOnes` should yield the two results `[]` and `[1, 1, 1]`, because we share the nondeterministic decision in `ones` across its own recursive call. This result is produced by both of the Curry Compilers *KICS2* and *PAKCS*. For our plugin, however, transforming `threeOnes` to behave semantically correct is quite challenging. To see the problem, let us consider the lifted version of the program without any explicit sharing.

```
threeOnes :: Nondet (ListND Int)
threeOnes = take >>= λf1 → f1 3 >>= λf2 → f2 ones
  where ones:: Nondet (ListND Int)
        ones = (?) (return NilND) (return (ConsND (return 1) ones))
```

Our plugin would now have to explicitly share the occurrences of `ones` in both the body of `threeOnes` and `ones` itself. However, we can only share an identifier after it has been defined, which prevents us from using the shared variable in the defintion of the function itself.

```
threeOnes :: Nondet (ListND Int)
threeOnes = share ones >>= λones' → take >>= λf1 → f1 3 >>= λf2 → f2 ones'
  where
    ones :: Nondet (ListND Int)  -- ones' not known in this defintion.
    ones = (?) (return NilND) (return (ConsND (return 1) ones'))
```

The example above is adapted from a paper by Christiansen, Seidel, and Voigtländer [2011], where they discuss the problems of defining a suitable operational semantics for recursive let expressions in the appendix A. Although the two Curry compilers *KICS2* and *PAKCS* produce the same results for the evaluation of `threeOnes`, there are programs that behave differently between both compilers. There is no obvious reason why our plugin should not be able to correctly transform at least each example that works with *KICS2*, because both the plugin and compiler work quite similarly. It might help to look at the differences between code compiled by the plugin and *KICS2* to figure out how to improve our semantic transformation, but this is left for future work. Adding another sharing operator for recursive definitions based on a fixed-point computation similar to the type class `MonadFix` might help to solve this problem in our transformation.

Note that sharing in let expressions is not implemented incorrectly in general. A good rule of thumb is that as long as a let expression can be decomposed into several nested lets, sharing across the nested bindings is possible. Internally, our plugin actually performs this splitting as part of the lifting transformation we described in

Section 4.1. There is only one problem with sharing and pattern bindings that was mentioned in Section 4.2.3 already: We do not perform any sharing across all variables introduced by a pattern binding. Although sharing should be possible to implement for non-recursive pattern bindings, our implementation conservatively puts all generated selector functions for pattern bindings in the same binding group, which prevents us from splitting the definitions like we did in the example above. GHC provides us with the information whether or not a binding is (mutually) recursive; however we just did not have the time for a more complex implementation.

## 4.4. Type Classes

After GHC has completed type checking the source code, every type class will be associated with a dictionary data type declaration and every class instance with a function binding that uses the corresponding dictionary constructor. This has two consequences for our plugin:

1. We have to lift every class declaration and its dictionary data type definition.
2. We have to lift the dictionary function and the corresponding class method implementations for every instance declaration.

### 4.4.1. Lifting Type Classes

The type constructor of a class declaration contains the dictionary data type declaration and some information on the type and name of any method or super class selector functions. The method selector functions have exacly the same name and type as the class method they correspond to. They are written in the source code by the user and not inserted by GHC, whereas super class selectors are used by GHC's constraint solver to extract a super class dictionary from a given class dictionary. A super class selector function that extracts a class `Super` $v_1 \ldots v_n$ from another class `MyClass` $v_1 \ldots v_n$ has the type `Super` $v_1 \ldots v_n \Rightarrow$ `MyClass` $v_1 \ldots v_n$. When lifting selector functions, we use our ordinary lifting $[\![\cdot]\!]$ for types for method selectors, whereas super class selectors are not lifted but use the type constructor replacement $[\![\cdot]\!]^r$ instead.

**Class Dictionaries** The associated dictionary data type definition for a class declaration contains exactly one constructor with an argument for every super class and class method in the order of their ocurrences. This can be seen in the following example.

**Example 4.4.1.** *Defining the type class* `Monad` *introduces the following internal data type defintion.*

```
data Monad m = C:Monad         -- ^ Internal constructor name
  (Applicative m)             -- ^ Applicative super class
  (m a → (a → m b) → m b) -- ^ (>>=)
  (m a → m b → m b)         -- ^ (>>)
  (a → m a)                   -- ^ return
```

```
-- Original class defintions
class MyEq a where
  equal :: a → a → Bool

class MyEq a ⇒ MyOrd a where
  lessOrEqual :: a → a → Bool

-- Original class constructors
newtype MyEq  a = C:MyEq           (a → a → Bool)
data    MyOrd a = C:MyOrd (MyEq a) (a → a → Bool)

-- lifted class constructors, not renamed because they replace the originals
newtype MyEq  a = C:MyEq           (Nondet (a :→ a :→ Bool))
data    MyOrd a = C:MyOrd (MyEq a) (Nondet (a :→ a :→ Bool))
```

**Figure 4.14.:** Lifting of class dictionary declarations

Although we do not want to lift the types of super class arguments, the type lifting function used for the lifting of ordinary constructor declarations also achieves the correct transformation for dicitionary constructors. Even though the type of a super class selector is of kind `Constraint`, which does not appear in value constructors of a data type definition, the transformation $[\![\cdot]\!]$ does not perform its lifting if the type is a constraint. This exception leads to a correct treatment of super class selectors in dictionary constructor types that can be seen in Figure 4.14.

**Single-Argument Type Classes** There is only one small problem for classes with exacly one super class or class method: GHC optimizes dictionaries for these kind of classes by using a newtype instead of a data type definition. Because of the special semantics for newtype constructors, their parameter is not lifted as outlined in Section 4.1.1. This requires us to add a special exception for the lifting of every newtype declaration that originated from a class declaration. For these newtypes, we use the normal lifting of data types as well. Additionally, a newtype class dictionary contains a coercion between its left and right side of the newtype defintion. This coercion has to be updated so that any optimizations performed by GHC on the Core intermediate language are correct. Figure 4.14 also shows the lifted version of a newtype class.

**Default Implementations** While the lifting of a default method does not differ from the lifting of an ordinary function, the presence of a default implementation for a class method poses some problems for its lifting. Although we normally keep the original version of each type constructor, we cannot keep any unlifted class declarations. If we were to keep them, we would have to remove their potentially nondeterministic default implementations. Even though this would be possible in theory, anyone who uses the unlifted original class might be confused by its lack of default implementations, leading to our decision to remove the unlifted class entirely.

### 4.4.2. Lifting Instances

After the type check, GHC has already created a dictionary function for each instance and renamed all instance functions with a unique name to avoid name clashes[11]. In Section 4.3.3, we have already seen the lifting and `Shareable` constraints of instance function types. However, we also have to lift the generated dictionary function and the actual implementation of instance functions. The following example shows the functions generated by GHC during type checking of an `Eq` instance for `Maybe`.

**Example 4.4.2.** *A dictionary function for an instance of* `Eq` *uses the corresponding class dictionary constructor* `C:Eq` *and applies it to all functions of the transformed instance.*

```
-- Original instance
instance Eq a ⇒ Eq (Maybe a) where
  (==) = (...)

-- Dictionary function
$fEqMaybe :: forall a. Eq a ⇒ Eq (Maybe a)
$fEqMaybe = C:Eq $c==_Maybe $c/=_Maybe

-- Instance functions without their implementation
$c==_Maybe :: forall a. Eq a ⇒ Maybe a → Maybe a → Bool
$c/=_Maybe :: forall a. Eq a ⇒ Maybe a → Maybe a → Bool
```

The instance functions are lifted by using the normal lifting for functions shown in Section 4.1.2, but dictionary functions need special treatment. Thankfully, we only need to update the types of the dictionary itself and any constructor or instance functions used in it, because the function does not have to be lifted into `Nondet`. After updating the types, we might need to pass any new `Shareable` constraints from the dictionary function to each instance function. This is quite simple, because each instance function in a dictionary requires the same type and evidence applications in the same order.

## 4.5. Built-In Type Definitions

Some of Haskell's type constructors like lists, tuples and numbers cannot actually be defined using a data type definition. Other type constructors are treated specially by GHC because they interact with the typing of syntactic constructs (i.e., `Num` for number literals or `Monad` for do-notation). Lastly, all type classes that are derivable by GHC have to be treated differently than other classes during the lifting of their derived instances. While we are not able to replace the type constructors that GHC uses or recognizes internally, our plugin can replace each occurence of a built-in type constructor with its own built-in version.

[11] GHC does not use the name of the type constructor to disambiguate instance functions, because that does not work for some language extensions. A generated `Unique` identifier is used instead.

**Loading Built-In Type Constructors** While talking about the lifting of data types in Section 4.1.1, we have already established that type constructors for data types, newtypes and type synonyms have to be replaced with their nondeterministic version throughout the plugin. By initializing the mapping used for the type replacement with values to replace built-in type constructors, we only have to provide a nondeterministic definition for each of GHC's special types that we care about. These manually lifted definitions are provided by a module in the plugin source code. Note that we currently only support 2-ary tuples, but other tuple sizes could be added.

**Deriving Built-In Type Classes** While the mechanism for replacing built-in type constructors described above would be enough to change the type class to its lifted version after deriving a class instance, the derived functions will always use and mention functions from Haskell's Prelude or base modules. This does not change even if the language extension `RebindableSyntax` is used. There is no way for us to influence GHC's deriving mechanism. However, we can manually edit the derived function while lifting it to replace each function from Haskell's Prelude with a built-in function from the plugin. Note that the type class `Read` is entirely unsupported by our plugin, because GHC uses parsers with higher-rank types for its `Read` class, contrary to what the Haskell 2010 standard says. By not exposing `Read` in our Prelude, we prevent the user from deriving it as well.

**Deriving `Enum`** Except for `Enum`, this ad-hoc function replacement works great. `Enum` instances generated by GHC use unboxed integers (`Int#`) as shown in the following example.

```haskell
-- Original datatype
data D = A | B | C deriving Enum

-- GHC's generated helper functions
$maxtag :: Int
$maxtag = I# 2#

$tag2con :: Int → D
$tag2con (I# a) = GHC.Prim.tagToEnum# a#

$con2tag :: D → Int#
$con2tag A = 0#
$con2tag B = 1#
$con2tag C = 2#

-- Exemplary succ function from Enum for datatype D
succ a = case $con2tag a of
      a# | a# == $maxtag (I# a#) → error "..."
         | otherwise             → $tag2con (I# a# + 1)
```

Our current lifting cannot handle unboxed types, which is a little problematic for `Enum`. Instead we detect if an instance of `Enum` is derived by checking for the usage

of $con2tag or other primitive functions. Any derived instance is lifted by using a specialized scheme where we compute the normal form of all parameters and convert them to their deterministic counterpart with `nf`. We can then use the original derived instance and convert the result back to the nondeterministic data type with `liftE`. For the function succ from above, we will generate the follwing lifted code.

```
succ = return $ λx → liftE (fmap succ' (nf x))
  where succ' a = case $con2tag a of -- succ' will be generated in-line
                    a# | a# == $maxtag (I# a#) → error "..."
                       | otherwise              → $tag2con (I# a# + 1)
```

## 4.6. Importing modules

After the plugin has lifted all definitions in a module, the ambient effect turns into an explicit effect. While it is necessary for the effect to be visible on the type level in order to generate type correct core code, this effect visiblilty is problematic when type checking code that uses imported definitions. All imported defintions have a lifted type, so GHC's type checker complains that imported defintions are used as if they are unlifted. Instead, GHC's type checker should treat them as unlifted. Furthermore, if the programmer uses any type class methods or writes any (type class) constraints, any evidenc generated by the constraint solver for these constraints will be wrong. Sometimes the constraints might be insoluble, even when they will be solveable after lifting is done completely.

At the start of the plugin project, we intended to use a plugin that runs after each interface has been loaded to un-lift any definition included in that interface. Unfortunately, modifying interfaces in an interface plugin is not supported by GHC. As an alternative, we opted to subvert GHC's type checker to allow using imported defintions.

### 4.6.1. Subverting GHC's Type Checker for Imports

One simple way to extend or modify GHC's type check is to implement a constraint solver plugin. Even though GHC's constraint solver only interacts with constraints, in Section 2.2.3 we have already mentioned that equality constraints are generated if the type checker fails to unify two types. The constraint solver plugin of our *Curry-Plugin* can intercept and modify these equalities to help the type checker in validating the type correctness of a given program. Our constraint solver plugin has four main goals:

1. It has to turn our effect type `Nondet` invisible by intercepting equality constraints and removing any occurrence of `Nondet` from both sides of the equality relation. The constraint solver uses the constraint produced by this transformation to (hopefully) show the equality bettween the unlifted left and right sides of the constraint.

2. The plugin also needs to automatically solve any constraints that mention the `Shareable` type class. There are a few different reasons why these constraints might be unsolvable for GHC:

   - The constraint might mention a type variable, but every `Shareable` constraint is added by the plugin during lifting. As a consequence, user-written code will never provide a given `Shareable` constraint for variables in the type of a function. For example, imagine that the imported function `toPair` has a lifted type `Shareable ⇒ Nondet (a :→ Tuple2ND a a)`. Then the following user-written function will fail to type check, because it is missing a `Shareable a` constraint that will not be added until after the type check.

     ```
     quadruple :: a → ((a, a), (a, a)) -- Unlifted type
     quadruple x = toPair (toPair x)     -- Missing Shareable a
     ```

   - The constraint might mention a type constructor that is defined in this module, but `Shareable` instances for local types are not created until after the type check. As an example, consider the following type `MyBool`. If we use the imported `toPair` function from above on a value of type `MyBool`, the type checker will complain about the missing `Shareable` instance, because the instance for `MyBool` does not exist during type checking.

     ```
     data MyBool         falsePair :: MyBool   -- Unlifted type
       = T | F           falsePair = toPair F  -- Missing Shareable MyBool
     ```

   - The constraint might mention a type constructor that is imported. Except for a few edge-cases, only the deterministic version of a type constructor, without a `Shareable` instance, is visible after importing. In the following example, we use the imported function `toPair` on a value of type `Maybe Bool`, but there is no `Shareable` instance for `Maybe`, only for its lifted counterpart `MaybeND`.

     ```
     noValuePair :: Maybe Bool     -- Unlifted type
     noValuePair = toPair Nothing -- Missing Shareable (Maybe Bool)
     ```

3. The second goal of our constraint solver plugin is to transform **given** type class constraints. Here, we sometimes have to treat the type constructor of the class constraint different than the types it is applied to. The applied types are transformed so that they only mention nondeterministic type constructors (if available), but for the type constructor of the class itself we might need to choose the deterministic version. While the choice between both class alternatives depends on a number of different factors, the plugin converts everything to the nondeterministic version, except if any of the type constructors has been defined in the current module.

4. The last goal of our constraint solver plugin is quite similar to the previous one. Any wanted class constraints also have to be transformed like above. The only difference is, that our plugin has to provide evidence (e.g., a suitable type class instance) for the transformation of a wanted constraint (see Section 2.2.3).

*We assume that* `repeat` *is imported and has type:* `Nondet (a0 :-> (ListND a0)))`. *According to GHC, the function below is incorrecly typed, because* `repeat` *is nullary and cannot be applied to an argument.*

```
replicate :: Int → a → [a]
replicate n a = take n (repeat xs)
  where
    take _ []      = []
    take n (x:xs)
      | n == 0     = []
      | n >  0     = x : take (n-1) xs
-- Could not match expected type ‘t → [a]’
-- with actual type ‘Nondet (a0 :→ (ListND a0))’
-- The function ‘repeat’ is applied to one argument,
-- but its type (...) has none.
```

*This definition produces the equality constraint*
`((t → [a]) ~# Nondet (a0 :→ (ListND a0)))`,
*which is transfomed to the solvable constraint*
`((t → [a]) ~# (a0 -> [a0]))`.

**Figure 4.15.:** Transforming an equality relation

**Turning `Nondet` invisible** The plugin transforms an equality constraint of the form $(\texttt{ty}_1 \texttt{ ~\# } \texttt{ty}_2)$ by removing any application of the `Nondet` type constructor and replacing any other type constructor with its deterministic version in both $\texttt{ty}_1$ and $\texttt{ty}_2$, resulting in $(\texttt{ty}'_1 \texttt{ ~\# } \texttt{ty}'_2)$. If none of the two types changed in the transformation, we cannot help GHC to solve this constraint and thus leave it unchanged. Otherwise, we now have to create the new constraint and a coercion for GHC to insert. Thankfully the coercion is irrelevant and will be removed by our lifting after the type check. For our new constraint it would be sufficient to only change the equality relation, but an equality constraint also contains information about its own “origin” and a description of the type error that lead to this constraint. If the new equality turns out to be unsolvable because of a type error in the user-written code, these informations are used by GHC to create the error message. Unless we clear the type error description and remove any mention of `Nondet` from the origin of the constraint, this error message contains confusing and outdated information. Figure 4.15 shows the transformation of an equality relation. As an alternative to “un-lifting” both sides of the equality, the plugin could also make sure that both sides contain a lifted type. GHC’s type check would still accept the same set of programs in this alternative implementation. However, any type errors generated for incorrect programs are less readable if we lift both sides, because the error message would mention the lifted types.

**Solving `Shareable` Constraints** If our constraint solver plugin encounters `Shareable` constraints, we mark them as solved. At this moment, we might not be able to produce any correct evidence for the constraint. Thankfully, all type class constraints, including

`Shareable`, will be solved during our main lifting phase again. Even if we create correct evidence for some cases, we would re-create it later on. To satisfy GHC's constraint solver, the plugin creates a dummy variable and tell GHC that this new variable contains the required type class dictionary. While the variable will always stay unbound and invalid, we know that the evidence will be discarded no matter what. Otherwise, one of the later transformations performed by GHC will complain about the variable not being in scope. One other thing to consider is, that after the lifting is done, every lifted data type and type variable always has a *Shareable* constraint, which is why solving the constraint later will never fail.

**Transforming other type class constraints** Although every type class constraint will be solved again later on, for any other type class than `Shareable`, we cannot just assume that it will be solveable. In the case that the programmer makes a mistake and the code is definitely incorrecly typed, we would like to produce a suitable error message. While transforming given and wanted type class constraints, we have to decide if they should be transformed to contain the lifted or unlifted class; a wrong desicion leads to a constraint that will never be solveable. There are a few conditions that make this decision possible:

- Any built-in class will never be a multi-parameter type class. Thus, if the number of types applied to the class is $\neq 1$, we know that the class is not built-in.
- According to Chapter 3 and Section 4.4, user-defined type classes do not have an unlifted version. Only built-in classes differentiate between a lifted and unlifted version of its class.
- Before the main lifing phase, all instances use the deterministic version of each type constructor.
- After the main lifting phase, all instances use the nondeterministic version of each type constructor in the instance definition. During constraint solving, this last condition is only relevant for imported instances, because they are the only instances that are already lifted during type checking of another module.

Both given and wanted constraints are handled with the following scheme: To determine the right course of action for our constraint transformation, we start by lifting the types that were applied to the class type constructor with our inner lifting $[\![\cdot]\!]^i$ and decide any further action based on the lifting result.

- If none of the applied types changed during this transformation, we know that they either contain only type variables and no type constructors, or that every one of those type constructors has no lifted version. Only data types that are defined in the current module and some built-in types do not have a lifted version. These data types can only have an instance that mentions their unlifted type constructor and an unlifted class, forcing us to unlift the class from the original constraint.

- If all of the types changed, we know that the constraint can only be solved if it mentions the lifted class.

- If the class is a multi-parameter type class and only some of the types changed, we thankfully know that there is no distinction between a lifted and unlifted class.

GHC does not guarantee us that our plugin never sees a constraint that was created by our plugin itself. To see that the lifting scheme above does not lead to infinite circles, one has to realize that we only ever "unlift" built-in type classes and only if the applied types did not change during lifting. If the types did not change during one invocation of the constraint solver plugin, they can never change in subsequent invocations, thus, any infinite circles are avoided.

As an example for the handling of constraints, consider the following data type `Nat` with a derived `Eq` instance and three variants of a `lookup` function.

```
data Nat = Zero | Succ Nat
  deriving Eq

lookup :: Eq k ⇒ [(k, a)] → Maybe a
lookup = (...)

lookupNat :: [(Nat, a)] → Maybe a
lookupNat = lookup

lookupMaybeNat :: [(Maybe Nat, a)] → Maybe a
lookupMaybeNat = lookup
```

To check if `lookupNat` is correctly typed, GHC has to solve the constraint `Eq Nat`, which is trivial. For `lookupMaybeNat`, however, we need to show that there is an instance of `Eq (Maybe Nat)`. Unfortunately, `Maybe` is imported and has no instance for `Eq`, only `MaybeND` has an instance for `EqND`. By transforming all type constructors in the wanted constraint to their nondeterministic counterpart, GHC can use the instance `EqND a ⇒ EqND (MaybeND a)` from the Plugin-Prelude to reduce the wanted constraint from `EqND (MaybeND Nat)` to `EqND Nat`. This new constraint is unsatisfiable again, because at this point in the compilation there is only a derived `Eq Nat` instance. As a consequence we un-lift the remaining constraint, so that GHC is able to fully verify the type of `lookupMaybeNat`.

One disadvantage of the current lifting scheme for type classes is that a user might be confronted with lifted type constructors in error messages. To fix the messages we would need to unlift any constraint that is unsolvable. While this detection is possible, GHC would pass us the new unlifted constraint in a subsequent constraint solver plugin invocation. Here, we would need some way to detect that the constraint originated from an unsolveable constraint to prevent us from lifting the constraint and falling into an endless circle. This problem remains to be solved in future work.

### 4.6.2. Marking Plugin Modules

To make sure that all definitions imported by the programmer are compatible with the lifting, we have to prevent any import of modules that did not use the plugin. By marking every plugin-compiled module with a module annotation pragma during the lifting, we are able to recognize compatible modules from the information contained in their interface files. GHC allows any expression with a `Data` instance as an annotation. The `Data` class provides a de- and serialization mechanism for data types. GHC uses these `Data` instances to serialize the annotation's value. The same restrictions as for spliced expressions in Template Haskell apply for all expressions inside annotations[12], because they both are evaluated at compile-time. For our plugin, however, none of these restrictions matter. The plugin uses the following annotation pragma for each module.

```
{-# ANN module Nondeterministic #-}
```

The expression `Nondeterministic` is a nullary constructor defined as follows.

```
data NondetTag = Nondeterministic
  deriving (Eq, Data)
```

We derive the `Data` instance for `NondetTag` by using `DeriveDataTypeable` as a language extension. Now the plugin can use the type `NondetTag` in annotations.

Checking all imports from the currently compiled module is done by the *Import Check* phase shown in the plugin overview at Figure 4.1 from the beginning of this chapter. The *Import Check* first gathers all imported modules and then checks if each of these imported modules contains a `Nondeterministic` annotation. For every invalid import we emit an error and abort the compilation if more than one error occurs. We do not check for this annotation in modules that are not directly imported by the current module for two reasons:

1. If a transitively imported module does not contain the required annotation, at least one of the imported modules must have failed to compile[13].

2. By only performing this shallow import check, we allow the user of our plugin to expose manually lifted definitions from plain Haskell modules into the plugin-compiled world. The user only has to add the required annotation pragma in the source code of the module to be imported. This mechanism is heavily used to expose built-in definitions in the new Prelude provided by our plugin.

We originally planned to also un-lift any imported definitions in the *Import Check*, but this is not possible with GHC's current implementation. GHC differentiates between modules from external packages and the so-called *home package*. While GHC hides external packages and their modules behind an `IORef` to only load them on demand, home package modules are not implemented with any statefulness and therefore

[12] Details are in the GHC wiki at https://gitlab.haskell.org/ghc/ghc/-/wikis/annotations.

[13] This simple check is sufficient even in the presence of cyclic module imports managed by `.hs-boot` files, although cyclic module dependencies are not supported by our plugin.

cannot be modified successfully. GHC uses the `HomePackageTable` to store modules from the home packace, except if GHC's one-shot compilation mode is used. In one-shot mode, GHC considers all modules to be from external packages and never saves them in the `HomePackageTable`. In contrast to compilation using GHC's simple make-style compilation with `--make` or interactive compilation with `GHCi`, one-shot compilation is significantly slower and cannot be used together with the tools *Stack* of *Cabal*. Thankfully, un-lifting of imported defintions becomes obsolete with the idea to use a constraint solver plugin to subvert GHC's type checker.

## 4.7. Further Implementation Details

This section explains some minor details that are not required to understand the main parts of our implementation.

**Record Selectors** After the type check has been performed by GHC, a selector function is generated for each record constructor. These selectors are also annotated wherever the corresponding record label is used in a record update or construction. When desugaring Haskell code to core, GHC will use the selector functions whenever a record update or construction is transformed. All selector functions are expected to be unary, otherwise an internal compiler error is thrown.

As a consequence, our plugin has to make sure that lifted selector functions can be applied to a value and produce a correctly lifted result. Our normal lifting for functions would transform a selector to a nullary function, so we use a different lifting for the definition of selectors that does not wrap the outer function layer as shown in the following example.

**Example 4.7.1.** *Consider a record defintion of the* `Identity` *data type.*

```
data Identity a = Identity { runIdentity :: a }
```

*GHC will automatically create a selector function for all record fields.*

```
runIdentity_selector :: Identity a → a
runIdentity_selector Identity { runIdentity = value } = value
```

*The lifting of a record selector is performed differently.*

```
runIdentity_selectorND :: Nondet (IdentityND a) → Nondet a
runIdentity_selectorND x = x >>= \case
  IdentityND { runIdentity = value } → value
```

Note that GHC is only able to use record patterns in the defintion of a selector, because record patterns are desugared without using the selector function itself. When our plugin lifts a record update, it also has to lift the annotated occurence of the selector function. Here we can treat selector functions like unary constructors, because they have the same type structure.

| Option name | Option description |
|---|---|
| `dump-original` | Dumps the type checked code before the plugin is run. |
| `dump-original-ev` | Dumps top-level evidence bindings before the plugin is run. |
| `dump-original-inst-env` | Dumps the instance environment before the plugin is run. |
| `dump-original-type-env` | Dumps the type environment before the plugin is run. |
| `dump-inst-env` | Dumps the instance environment after instance information is lifted. |
| `dump-pattern-matched` | Dumps the code after pattern matching has been compiled. |
| `dump-deriving-errs` | Dumps internal errors during deriving of internal classes. |

**Figure 4.16.:** Plugin-specific debug options

**Debugging the Plugin** There are several flags for the plugin that can be activated to output intermediate results of the plugin transformation. They complement GHC's pre-existing debug flags that are documented at Section 6.13 of the GHC User's Guide. The most commonly used flag when debugging the plugin is `-ddump-tc`, because it outputs the code after GHC has type checked it. Any activated plugin for the type checker is run before the output as well, so the flag can be used to get the final transformation output of the *Curry-Plugin*. Every plugin-specific debug option is shown in Figure 4.16. All of them have to be prefixed with `-fplugin-opt Plugin.CurryPlugin:` so that GHC knows to which plugin they belong.

**Testing the Plugin** The project for the plugin also contains a small test suite based on Cabal's *detailed-0.9* framework that can be used to run two different types of tests:

- Compilation tests that check if a given list of test modules can be compiled with the plugin. It is used to test the robustness of the plugin transformation and as a check that only plugin-compiled modules can be imported.
- Semantic tests that check if a given nondeterministic definition from a plugin-compiled module behaves as expected. All nondeterministic defintions are captured lazily to allow testing of infinite trees of nondeterministic choices.

Our test suite currently contains a small number of tests, but it could be expanded easily. With these tests in place, we were able to find a regression problem when we changed our approach for sharing polymorphic values as outlined in Section 4.6. The test were also helpful to locate a problem with too strict encapsulation in a previous implementation where we did not use the sharing library by Fischer, Kiselyov, and Shan [2011].

**Installing the Plugin** The plugin can be installed for GHC 8.10.1 by running `cabal install --lib curry-ghc-plugin` from the root of the project. Other compiler versions are incompatible. The plugin can also be used via Stack, where we even provide an example project to play around with in the sandbox subfolder of the project. By executing `stack repl sandbox`, one can load the example in a GHCi session. The plugin is not (yet) available on Hackage or Stackage.

**Error Messages** Most of the error messages a user will see are generated by GHC's parser, name resolution or type checker. Although our constraint solver plugin heavily modifies types to guide the compiler's type check, our implementation ensures that type errors are as readable as possible. In some cases, however, our plugin generates its own error messages.

1. When the *Curry-Plugin* detects an incompatible module among the imported modules (see Section 4.6), the plugin emits an error message that points to the problematic import declaration.

2. Whenever the monadic transformation of any syntactic element introduced by a language extension fails, our plugin generates a message that informs the user about the unsupported extension. Although we could check all enabled language extensions before trying to transform a module, we intentionally decided against this check and instead generate a message for the places where such an extension is used. By deferring the check to use-sites, we allow the user to enable and use any extensions, as long as they do not impact our transformation.

3. Although most bugs in our implementation are hopefully fixed, any error that we did not expect and cannot recover from is assumed to be a bug and reported to the user as a *panic* message. A *panic* is used by GHC for unexpected error messages that it cannot recover from.

In each case, we try to enrich every generated error message with a source location that points to the problematic part of the module.

# 5. Evaluation

In this chapter we will evaluate our approach to implement a Curry compiler as a GHC plugin. For all categories we will compare our plugin with Curry's two main compiler implementations: KICS2 and PAKCS.

## 5.1. Compilation Performance

To compile a module containing only a function to compute all permutations of a list, our plugin requires around $3.5s$. While this seems relatively high, most of the time is spent loading module interfaces from various packages. Without our plugin, GHC normally does not load the packages unless required, but with the plugin the compiler is forced to load them. If a different plugin-compiled module is loaded or when the permutations module is recompiled with `-fforce-recomp` in the same session, the compilation time drops to $0.5s$ where only $0.05s$ are spent in the type checking and lifting phase. It might be possible to prevent the high compilation time for the first module, because there is no obvious reason why the plugin forces all packages to be loaded. To put these times into perspective, the PAKCS and KICS2 compilers for Curry take approximately $2.4s$ to load a similar module. For more than a single module, our plugin outperforms both other compilers during compilation. This was expected, as PAKCS and KICS2 have to call more than a single executable/compiler for every module. While the time to lift modules does grow linearly with the size ($m$) of the module in general, looking up a type constructor might take a time of $\mathcal{O}(\log(n))$, where $n$ is the number of in-scope type constructors. This might lead to a time complexity of $\mathcal{O}(m \log(n))$ for a plugin invocation that will be added on top of GHC's complexity [1].

## 5.2. Execution Performance

The performance of plugin-compiled code depends heavily on the use of sharing and nondeterminism. Thus, we test variations of the following three programs.

```
test1 :: Int → Int   -- 1. No sharing, no nondeterminism
test1 n = foldr (+) 0 [1..n]

test2 :: Int → Int   -- 2. No sharing, with nondeterminism
test2 n = (foldr (?) 1 [2..n]) + 1
```

[1] We do not include the complexity of pattern matching in our calculation, because complex patterns would have to be desugared by GHC without the plugin.

```
test3 :: Int → [Int] -- 3. Sharing and nondeterminism
test3 n = permutations [1..n]
```

Of course, we have to wrap these functions with the correct encapsulation to test them in the *Curry-Plugin*. Note that `foldr` shares its first parameter, but sharing a function is almost a no-op and can be ignored. For the second test, we capture all results and print them to the console to force their evaluation. In the last test it is sufficient to print the number of permutations for a good measurement.

Both compilers and the plugin have similar run-times for all length variations of the first test, but PAKCS has an overhead of approximately $1s$. The other two tests are more interesting and Figure 5.1 contains a graph of the execution times for all compilers of both tests. We can see that the plugin outperforms KICS2 and PAKCS in the nondeterminism test, but further investigation revealed that a small portion of the plugin's advantage came from a smaller overhead at performing IO.

As soon as *Curry-Plugin* has to share values between computations in the permutations test, its performance drops significantly compared to KICS2 and PAKCS. Profiling the test program with GHC confirms that our plugin spent one third of its time traversing lists for their `Shareable` instance with `<*>`. An additional 20% are used for other parts of the implementation of `share`. Thankfully, the time to look up shared values in the state that is used by the sharing implementation of Fischer, Kiselyov, and Shan [2011] seems to be negligible. One thing to note is that the `Shareable` instance for lists is hand-written and does not use our generic deriving mechanism. Other performance tests show that sharing via generically derived instances is not noticeably slower than hand-written instance because of excessive inlining performed by GHC. We will discuss some future work to increase the plugin's performance in the context of sharing in Section 6.3

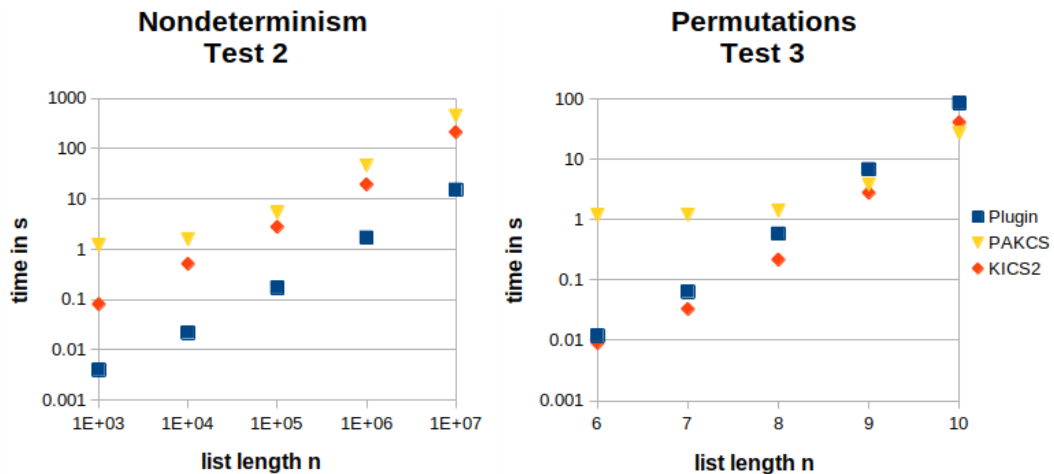


**Figure 5.1.:** Execution time comparisons of both compilers and the plugin

## 5.3. Language Features

In its current form, the Curry plugin has two major feature restrictions when compared to both other Curry compilers:

1. Our plugin does not support the declaration of free variables (see Chapter 3).
2. Our plugin does not support sharing of choices in recursive local declarations and pattern bindings (see Section 4.3.4).
3. Our plugin does not support functional patterns (see [Antoy and Hanus, 2006]).

While the first restriction can be mitigated by using generator functions [Hanus and Teegen, 2020], the second restriction is more severe and requires future work. However, I conjecture that there are few applications for sharing within recursive bindings.

On the positive side, my plugin already supports a lot of language extensions "for free", that Curry is currently lacking. Aside from a lot of minor extensions like `LambdaCase`, we also support `MultiParamTypeClasses`, which are quite handy and have been used in the context of monad transformers by M. P. Jones [1995b]. A full list of language extensions and their level of support can be seen in Appendix C. One of the advantages of our plugin is that the list of supported extensions can be increased relatively easy, because most of the heavy-lifting is done by GHC already. If we were to adapt such an extension into one of the other Curry-Compilers, we would need significantly more lines of code – and more code leaves more room for bugs.

Aside from language extensions, the possibility to integrate "Curry-Code" within Haskell with our plugin makes developing applications that use nondeterminism much easier because of the better availability of tooling and libraries in the Haskell ecosystem.

## 5.4. Maintainability

On the topic of maintainability, we are interested in the challenges of fixing bugs, implementing new features and upgrading to new GHC versions.

**Maintaining KICS2 and PAKCS** The experience from recent years has shown that the common frontend for KICS2 and PACKS is easy to port to new GHC versions. By recently switching the build tool for the frontend to Stack, compilation is also independent from the build systems configuration. The backend of KICS2 however had some difficulties with new GHC flags and optimizations in the past and sometimes does not work with new versions. Debugging these problems is usually possible, but time consuming. Implementing new features for both Curry compilers will almost surely require bigger changes to the common frontend, sometimes without changing the intermediate language called *FlatCurry*. This makes it possible for students to work on the compiler as a part-time job or as part of their thesis. The current code base seems relatively easy to work with.

**Maintaining the Plugin** Our Curry plugin also uses Stack as a build tool, but upgrading its code-base to a new GHC version requires some effort due to the tight integration into the compiler. Our plugin project started with GHC version 8.6.5 and was later moved two versions ahead to GHC 8.10.1, because a specific bug fix from the newer version was required. Note that GHC does not use strict semantic versioning, instead the second version number increases between each release, where odd numbers are reserved for development releases. Major version steps are reserved for big new features. All changes that were required for the upgrade of our plugin to GHC 8.10.1 were implemented within one day, even though "visible dependent quantification" was added by Scott [2019] and required some changes to the type system. Thankfully GHC's internal type system was only affected slightly.
In the near future, GHC 9.0 will be released with the new (optional) linear type system. Linear types required a lot of changes to GHC's type checker and inference, but the internal types will be changed only slightly again. According to a recent talk by Simon L. Peyton Jones and Ben Gamari at the 2020 Haskell Implementors Workshop[2], GHC 9.0 will also contain a lot of changes that might simplify our plugin implementation quite a bit, but that is just speculation for now.

To conclude the section about maintainability, our plugin requires more work to port it to new GHC versions than current compilers. However it also profits a lot more from an upgrade to a new GHC, because a new version gives us the opportunity to implement new language extensions with much less additional effort.

[2]https://icfp20.sigplan.org/details/hiw-2020-papers/12/GHC-Status-Update

# 6. Conclusion

This chapter summarizes our results, shows similarities or differences to other related work and gives an outlook on possible future work.

## 6.1. Summary and Results

In this thesis we have implemented a compiler based on a GHC plugin for the Curry programming language.

When activated, our plugin performs a monadic lifting on the source code of a module to integrate nondeterminism as an ambient effect into an existing Haskell module. To correctly model call-time choice semantics, we explicitly share variables in our monadic lifting. By using quantified constraints, we could provide the required type classes for our implementation even in the context of types with higher-kinded type variables. As a consequence, *Curry-Plugin*'s semantic transformation is enough to achieve a Curry-style functional-logic language. Our implementation also serves as a proof-of-concept for a bug fix in the KICS2 compiler (KICS2 Issue #28[1]).

With *Curry-Plugin* we have reached our goal in supporting the whole Haskell 2010 standard, including type classes and deriving mechanisms. By using a constraint solver plugin, we achieve an ambient effect that remains hidden from the user, even when importing definitions from within a plugin-compiled module. Although the plugin could be modified to be used as a compiler, it is more convenient to embed plugin-compiled code within native Haskell as some form of (embedded) domain specific language. The implementation of our plugin as described in this thesis is easy to modify to support other effects and language extensions.

We have also provided a comparison of the plugin with the two main Curry compiler implementations to show the feasibility and strength of our approach, especially the easy integration of new language extensions that are not supported by other Curry-Compilers.

## 6.2. Related Work

Current implementations of Curry compilers generate Haskell code that is then further processed by GHC for the remaining transformation to machine code [Braßel et al., 2011]. The generated Haskell code implicitly uses a monadic lifting similar to the one used in *Curry-Plugin*, but fully in-lines the nondeterminism effect into each datatype.

[1] `https://git.ps.informatik.uni-kiel.de/curry/kics2/-/issues/28`

In contrast to our *Curry-Plugin* they have to implement a parser, type checker and other parts of a compiler from scratch.

To reduce the code that has to be written for the compiler of the Eden language, Loogen, Ortega-Mallén, and Peña-Marí [2005] base their work on a fork of GHC to avoid re-implementing any features. However, keeping up with GHC's development using this approach is still problematic.

GHC plugins have been used for a variety of different tools [Pickering, Wu, and Németh, 2019]. One example that is most similar to *Curry-Plugin* is a core-plugin from Elliott [2017] that implements a full code transformation based on categories to allow different interpretations of the same code. Their goal, however, is not to implement a different programming language. Another example for a plugin that extends GHC's type check is the plugin version of *Liquid Haskell* by [Vazou et al., 2014], a tool to introduce refinement types into Haskell. Their plugin collects information about a module from its desugared Core code to generate constraints that are then solved by the pre-existing Liquid Haskell SMT-based implementation. The *type-nat-solver* by Diatchki [2015] extends GHC's constraint solver with a plugin to solve type-level constraints with booleans and natural numbers using an SMT-Solver as the backend as well.

The monadic lifting used by our plugin is based on work done by Wadler [1990], while the sharing implementation for call-time choice semantics is based on a paper by Fischer, Kiselyov, and Shan [2011] that shows how lazy nondeterministic programming can be achieved in a pure functional way. Their implementation however requires the user to write all code in a monadic fashion.

## 6.3. Future Work

While there are still a few bugs left to fix in our implementation (e.g., tuples with arity $> 2$ and deriving for `Read`), we propose a few open questions for further research. Our future work can be put into three distinct categories:

1. Improving the transformation
2. Implementing support for more language extensions
3. Generalizing the plugin for more effects.

The first point mainly includes future optimizations to reduce the overhead of lifting and sharing of computations. Our last point contains ambitious steps to turn the plugin into a compiler for a language with a first-class effect system, similar to languages like *Eff* by Bauer and Pretnar [2015].

### 6.3.1. Improving the Transformation

We have already mentioned our problems with implementing a correct semantic transformation for effectful recursive bindings in Section 4.3.4. While a good, composable operational semantic is hard to define for such bindings, we need to implement a so-

lution that is at least as good as the transformation achieved by KICS2. Adding a new sharing-operator based on a monadic fixed-point computation might be a possible solution, but so far it has not been tested.

On the topic of performance, we have seen that sharing computation slows us down significantly, even if compared to KICS2. The advantage that KICS2 has over our plugin is, that sharing only occurs for nondeterministic values by labeling choices instead of blindly sharing all values, even if they are effect-free. To improve the performance of our plugin, we could skip sharing values if we know them to be effect-free, but that requires a complicated effect analysis. Implementing such an analysis and optimizing the transformation based on the effectfulness of values is one point for future research.

For a lot of inductively defined functions, `share` might be invoked multiple times with the same values. Consider the following lifted variant of `insert` that is used when computing all permutations of a list throughout this thesis.

```
insert :: Shareable Nondet a ⇒ Nondet (a :→ ListND a :→ ListND a)
insert = return $ λx' → share x' ≫= λx → return $ λxs → xs ≫= \case
   Nil         → return (Cons x' (return Nil))
   Cons y' ys' → share y' ≫= λy → share ys' ≫= λys →
     (return (Cons x (return (Cons y) (return Nil))))
     ? (return (Cons y (insert ≫= λf1 → f x ≫= λf → f2 ys)))
```

Here we use `share` multiple times throughout all recursive invocations of `insert`: Once for each sub-list (tail) of the second parameter (`zs`), once for each head of `zs` and once for the first argument of `insert`. If we also count the sharing that is required to share the list components when invoking `share` on `zs`, each element in `zs` will be shared one more time than its predecessor. This leads to $\mathcal{O}(\text{length}(\texttt{zs})^2)$ invocations of `share`, which is obviously too much. If we know that the full list `zs` will be demanded eventually, then we could compute its normal form to trigger all sharing and then continue with a sharing-free version of insert. This optimization would effectively implement a call-by-value semantics and is not possible in general for call-by-need evaluation.

While evaluating profiling results obtained with the benchmarks in Chapter 5, we also noticed that GHC seems to have trouble optimizing our generated code. Even after increasing GHC's in-lining threshold and the number of optimization passes, and even after adding some rewrite rules, performance increased only slightly. As the last point in the category of future performance improvements, we want to investigate how to enable GHC to optimize plugin-generated code even further.

### 6.3.2. Language Extensions

GHC currently has over 100 language extensions, ranging from simple syntactic extensions like `LambdaCase` to advanced extensions like `TypeFamilies`. Our plugin currently supports only a subset of all extensions as seen in Appendix C.

**Datatype Extensions** With the current deriving scheme for `Shareable` for data types we cannot support data type extensions like `GADTs`, because `Generic` is not derivable for

such data types. We could change our deriving scheme by generating all `Shareable` instances with the plugin without using GHC's `Generic`. That would allow us to support at least `GADTs`, because their instances are still easy enough to generate. To support unlifted types (e.g., `UnboxedSums`, `UnliftedTuples`) we could adapt our lifting to behave differently if a value has an unlifted kind. Something similar has been done by Downen et al. [2020] to encode the correct calling convention for a function or value on the kind-level.

**Type System Extensions** GHC's full type system is incredibly rich and some of its extensions are not entirely compatible (see GHC bug report #17295[2]). While we think that support for higher-rank types and type families should be possible to implement, their implementation might require significant changes to the plugin. If we look even further, it is unclear how data type promotion and polymorphic kinds should interact with plugin-lifted definitions. A promoted data type should probably not be lifted by the plugin, but we have not thought about all interactions with the plugin. We instead leave this question open for future work.

**Other extensions** There are a lot of extensions that might work out-of-the-box that have not been tested with the plugin at all, so we have a lot to do in the future on that front. We also know that overloading of syntactic constructs with `RebindableSyntax` can be supported even with the plugin enabled. In fact, a small part of it should work already. While that is a lesser-used extension in Haskell, it might see more use in the future with the arrival of linear types, because that extension often requires importing linear versions of existing definitions like a linear monad type class. Without `RebindableSyntax` or the planned `RebindableDo`, do-notation would be unusable with linear types.

### 6.3.3. Generalization for other Effects

We have already mentioned in Section 3.2, that the monadic lifting performed in our plugin is completely orthogonal to the nondeterminism effect we use. We have already shown that our plugin can be adapted to a probabilistic effect by changing approximately 500 lines of code. We could also implement a plugin for reactive programming Van Der Ploeg [2013] to write programs in a reactive style without explicitly using a monad.

But forking and changing the plugin for every effect that we want to use seems unnecessary when the lifting and effect are orthogonal to each other. Our goal for the future is to let the user specify an effect and corresponding handlers in a module to be used with the plugin. That would enable our plugin to transform GHC to a compiler for a language with effects as a first class language feature, similar to the mentioned language *Eff* by Bauer and Pretnar [2015]. One of the challenges when implementing this generalization will be to provide an effect-independent Prelude to

[2] https://gitlab.haskell.org/ghc/ghc/issues/17295

be used independently of the effect. We might also reconsider our approaches to some of the previously mentioned future work, especially the interaction between language extensions and our plugin will be more deliberate if we consider arbitrary monadic effects. As another step, we could adapt the plugin to let the user choose between evaluation strategies like call-by-value and call-by name.

# Bibliography


Alqaddoumi, Abdulla et al. (2010). “The pull-tab transformation”. In: *Proc. of the Third International Workshop on Graph Computation Models*, pp. 127–132.

Antoy, Sergio and Michael Hanus (2006). “Declarative Programming with Function Patterns”. In: *Logic Based Program Synthesis and Transformation*. Ed. by Patricia M. Hill. Berlin, Heidelberg: Springer, pp. 6–22. DOI: 10.1007/11680093_2.

Bauer, Andrej and Matija Pretnar (2015). “Programming with algebraic effects and handlers”. In: *Journal of Logical and Algebraic Methods in Programming* 84.1, pp. 108–123. DOI: 10.1016/j.jlamp.2014.02.001.

Braßel, B. et al. (2011). “KiCS2: A New Compiler from Curry to Haskell”. In: *Proceedings of the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011)*. Springer LNCS 6816, pp. 1–18. DOI: 10.1007/978-3-642-22531-4_1.

Cassel, John (2014). “Probabilistic Programming with Stochastic Memoization Implementing Nonparametric Bayesian Inference”. In: *The Mathematica Journal* 16. DOI: 10.3888/tmj.16-1.

Christiansen, Jan, Daniel Seidel, and Janis Voigtländer (2011). “An adequate, denotational, functional-style semantics for Typed FlatCurry”. In: *International Workshop on Functional and Constraint Logic Programming*. Springer. Berlin, Heidelberg: Springer, pp. 119–136. DOI: 10.1007/978-3-642-20775-4_7.

Diatchki, Iavor S. (2015). “Improving Haskell Types with SMT”. In: *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*. Haskell ’15. Vancouver, BC, Canada: Association for Computing Machinery, pp. 1–10. DOI: 10.1145/2804302.2804307.

Downen, Paul et al. (2020). “Kinds Are Calling Conventions”. In: *Proc. ACM Program. Lang.* 4. DOI: 10.1145/3408986.

Elliott, Conal (2017). “Compiling to Categories”. In: *Proceedings of the ACM on Programming Languages* 1.ICFP. DOI: 10.1145/3110271.

Fischer, S., O. Kiselyov, and C. Shan (2011). “Purely functional lazy nondeterministic programming”. In: *Journal of Functional programming* 21.4&5, pp. 413–465. DOI: 10.1017/S0956796811000189.

Hanus, Michael (2019). *Declarative Programming Languages – Lecture Notes (in German)*. URL: https://www-ps.informatik.uni-kiel.de/~mh/lehre/dps19/.

Hanus, Michael and Finn Teegen (2020). “Adding Data to Curry”. In: *Declarative Programming and Knowledge Management*. Ed. by Petra Hofstedt et al. Springer International Publishing, pp. 230–246. DOI: 10.1007/978-3-030-46714-2_15.

Hanus (Ed.), Michael (2016). *Curry: An Integrated Functional Logic Language (Vers. 0.9.0)*. Available at http://www.curry-language.org.

Hennessy, M. C.B. and E. A. Ashcroft (1977). “Parameter-Passing Mechanisms and Nondeterminism”. In: *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*. STOC ’77. Boulder, Colorado, USA: Association for Computing Machinery, pp. 306–311. DOI: 10.1145/800105.803420.

Hindley, R. (1969). “The Principal Type-Scheme of an Object in Combinatory Logic”. In: *Transactions of the American Mathematical Society* 146, pp. 29–60. DOI: 10.2307/1995158.

Jeuring, Johan et al. (2009). “Libraries for Generic Programming in Haskell”. In: *Advanced Functional Programming: 6th International School, AFP 2008, Heijen, The Netherlands, May 2008, Revised Lectures*. Ed. by Pieter Koopman, Rinus Plasmeijer, and Doaitse Swierstra. Berlin, Heidelberg: Springer, pp. 165–229. DOI: 10.1007/978-3-642-04652-0_4.

Jones, Mark P. (1995a). “A system of constructor classes: overloading and implicit higher-order polymorphism”. In: *Journal of Functional Programming* 5.1, pp. 1–35. DOI: 10.1017/S0956796800001210.

Jones, Mark P. (1995b). “Functional programming with overloading and higher-order polymorphism”. In: *Advanced Functional Programming*. Ed. by Johan Jeuring and Erik Meijer. Berlin, Heidelberg: Springer, pp. 97–136. DOI: 10.1007/3-540-59451-5_4.

Loogen, Rita, Yolanda Ortega-Mallén, and Ricardo Peña-Marí (2005). “Parallel functional programming in Eden”. In: *Journal of Functional programming* 15.3, pp. 431–475. DOI: 10.1017/S0956796805005526.

Magalhães, José Pedro et al. (2010). “A Generic Deriving Mechanism for Haskell”. In: *SIGPLAN Not.* 45.11, pp. 37–48. DOI: 10.1145/2088456.1863529.

Matthes, Jan-Hendrik (2019). “Extending Curry with higher-rank polymorphism (in German)”. MA thesis. Germany: CAU Kiel.

Milner, Robin (1978). “A theory of type polymorphism in programming”. In: *Journal of Computer and System Sciences* 17.3, pp. 348–375. DOI: 10.1016/0022-0000(78)90014-4.

Najd, Shayan and Simon Peyton Jones (2017). “Trees that Grow”. In: *J. UCS* 23.1, pp. 42–62. DOI: 10.3217/jucs-023-01-0042.

Odersky, Martin and Konstantin Läufer (1996). “Putting Type Annotations to Work”. In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’96. St. Petersburg Beach, Florida, USA: Association for Computing Machinery, pp. 54–67. DOI: 10.1145/237721.237729.

Pickering, Matthew, Nicolas Wu, and Boldizsár Németh (2019). “Working with Source Plugins”. In: *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell*. Haskell 2019. Berlin, Germany: Association for Computing Machinery, pp. 85–97. DOI: 10.1145/3331545.3342599.

Plotkin, Gordon and Matija Pretnar (2009). “Handlers of Algebraic Effects”. In: *Programming Languages and Systems*. Ed. by Giuseppe Castagna. Berlin, Heidelberg: Springer, pp. 80–94. DOI: 10.1007/978-3-642-00590-9_7.

Scott, Ryan (2019). *Visible Dependent Quantification in Haskell - Ryan Scott*. URL: https://ryanglscott.github.io/2019/03/15/visible-dependent-quantification-in-haskell/.

Teegen, Finn (2016). "Extending Curry with type classes and type constructor classes (in German)". MA thesis. Germany: CAU Kiel.

The Rust Team (2016). *Rust Project - Frequently Asked Questions*. URL: https://prev.rust-lang.org/en-US/faq.html.

Van Der Ploeg, A. J. (2013). "Monadic Functional Reactive Programming". In: *Proceedings of the ACM SIGPLAN Haskell Symposium*. Ed. by C Shan. DOI: 10.1145/2578854.2503783.

Vazou, Niki et al. (2014). "Refinement Types for Haskell". In: *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*. ICFP '14. Gothenburg, Sweden: Association for Computing Machinery, pp. 269–282. DOI: 10.1145/2628136.2628161.

Wadler, Philip (1987). "Efficient Compilation of Pattern-Matching". In: *The Implementation of Functional Programming Languages*. Ed. by Simon L. Peyton Jones. Prentice-Hall, pp. 78–103. DOI: 10.1016/0141-9331(87)90510-2.

Wadler, Philip (1990). "Comprehending Monads". In: *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*. LFP '90. Nice, France: Association for Computing Machinery, pp. 61–78. DOI: 10.1145/91556.91592.

# A. Transformation of a Small Example

```
-- code after pre-processing
newtype Identity a = Identity a
data    Pair     a = Pair a a

mapIdentity :: (a → b) → Identity a → Identity b
mapIdentity = λf → λv → case v of
  Identity a → Identity (f a)

mapBoth :: (a → b) → Pair a → Pair b
mapBoth = λf → λv → case v of
  Pair a1 a2 → Pair (f a1) (f a2)

-- code after lifting
newtype IdentityND a = IdentityND a
data    PairND     a = PairND (Nondet a) (Nondet a)

mapIdentity :: Nondet ((a :→ b) → IdentityND a :→ IdentityND b)
mapIdentity = return (λf → return (λv → v >>= \case
  IdentityND a' → let a = return a' in
    (return $ λx → fmap IdentityND x) >>= λf1 → f1
      (f >>= λf2 → f2 a)))

mapBoth :: Nondet ((a :→ b) → PairND a :→ PairND b)
mapBoth = return (λf' → share f' >>= λf → return (λv → v >>= \case
  PairND a1 a2 → (return $ λx1 → return $ λx2 → return (Pair x1 x2)) >>=
    λf1 → f1 (f >>= λf2 → f2 a1) >>= λf2 → f2 (f >>= λf2 → f2 a2)))
```

# B. Generic Implementation of Shareable

```
class Shareable m a where
  shareArgs :: (Monad n) ⇒

  default shareArgs :: (Gen.Generic a, ShareableGen m (Gen.Rep a), Monad n) ⇒
    (forall b. (Shareable m b ⇒ m b → n (m b))) → a → n a
  shareArgs f a = Gen.to <$> shareArgsGen f (Gen.from a)

class ShareableGen m f where
  shareArgsGen :: (Monad n) ⇒
    (forall b. (Shareable m b ⇒ m b → n (m b))) → f x → n (f x)

instance (Monad m) ⇒ ShareableGen m Gen.V1 where
  shareArgsGen _ v = case v of
    -- No value of this type can exist

instance (Monad m) ⇒ ShareableGen m Gen.U1 where
  shareArgsGen _ = return

instance (Monad m, ShareableGen m f, ShareableGen m g) ⇒
  ShareableGen m (f Gen.:+: g) where
    shareArgsGen f (Gen.L1 x) = Gen.L1 <$> shareArgsGen f x
    shareArgsGen f (Gen.R1 x) = Gen.R1 <$> shareArgsGen f x

instance (Monad m, ShareableGen m f, ShareableGen m g) ⇒
  ShareableGen m (f Gen.:*: g) where
    shareArgsGen f (x Gen.:*: y) =
      (Gen.:*:) <$> shareArgsGen f x <*> shareArgsGen f y

-- | This instance overlaps the next instance.
--   Any lifted type defined by a data declaration uses this instance,
--   the other instance is used for lifted newtypes.
instance {-# OVERLAPPING #-} (Monad m, Shareable m b)
  ⇒ ShareableGen m (Gen.K1 i (m b)) where
    shareArgsGen f (Gen.K1 x) = Gen.K1 <$> f x

instance {-# OVERLAPPABLE #-} (Monad m, Shareable m c)
  ⇒ ShareableGen m (Gen.K1 i c) where
    shareArgsGen f (Gen.K1 x) = Gen.K1 <$> shareArgs f x

instance (Monad m, ShareableGen m f) ⇒ ShareableGen m (Gen.M1 i t f) where
  shareArgsGen f (Gen.M1 x) = Gen.M1 <$> shareArgsGen f x
```

# C. List of Language Extension Support

The following lists shows all language extensions for GHC 8.10.1 and if they are supported by the plugin. For extensions that are marked as supported, there is no reason for them not to work. Either because they are fully desugared by GHC before the type checker plugin is run, or because explicit support for the extension was implemented for the plugin. Not that not all of the extensions marked as supported have been tested, there is rather no reason for them not to work. Extensions that we were unsure about or know that they are incompatible with the plugin have been marked as unsupported. An unsupported extension sometimes contains a note on how difficult they would be to implement.

| **Extension** | **Support Status** |
|---|---|
| `AllowAmbiguousTypes` | Supported |
| `ApplicativeDo` | Unsupported, but easy to implement |
| `Arrows` | Unsupported |
| `BangPatterns` | Unsupported, but very easy to implement |
| `BinaryLiterals` | Supported |
| `BlockArguments` | Supported |
| `CApiFFI` | Unsupported |
| `ConstrainedClassMethods` | Supported |
| `ConstraintKinds` | Unsupported |
| `CPP` | Supported |
| `DataKinds` | Unsupported |
| `DatatypeContexts` | Unsupported |
| `DefaultSignatures` | Supported |
| `DeriveAnyClass` | Supported |
| `DeriveDataTypeable` | Unsupported |
| `DeriveFoldable` | Unsupported |
| `DeriveFunctor` | Supported |
| `DeriveGeneric` | Unsupported |
| `DeriveLift` | Unsupported |
| `DeriveTraversable` | Unsupported |
| `DerivingStrategies` | Supported |
| `DerivingVia` | Supported |
| `DisambiguateRecordFields` | Supported |
| `DuplicateRecordFields` | Supported |
| `EmptyCase` | Supported |

| | |
|---|---|
| `EmptyDataDecls` | Supported |
| `ExistentialQuantification` | Unsupported |
| `ExplicitForAll` | Supported |
| `ExplicitNamespaces` | Supported |
| `ExtendedDefaultRules` | Unsupported |
| `FlexibleContexts` | Supported |
| `FlexibleInstances` | Supported |
| `ForeignFunctionInterface` | Unsupported |
| `FunctionalDependencies` | Supported |
| `GADTs` | Unsupported |
| `GADTSyntax` | Supported |
| `GeneralisedNewtypeDeriving` | Supported |
| `HexFloatLiterals` | Supported |
| `ImplicitParams` | Unsupported |
| `ImplicitPrelude` | Required |
| `ImpredicativeTypes` | Unsupported |
| `IncoherentInstances` | Supported |
| `InstanceSigs` | Supported |
| `InterruptibleFFI` | Unsupported |
| `KindSignatures` | Supported |
| `LambdaCase` | Supported |
| `LiberalTypeSynonyms` | Unsupported |
| `MagicHash` | Supported |
| `MonadComprehensions` | Unsupported, but almost implemented |
| `MonadFailDesugaring` | Supported |
| `MonoLocalBinds` | Supported |
| `MonomorphismRestriction` | Supported |
| `MultiParamTypeClasses` | Supported |
| `MultiWayIf` | Supported |
| `NamedFieldPuns` | Supported |
| `NamedWildCards` | Supported |
| `NegativeLiterals` | Supported |
| `NPlusKPatterns` | Unsupported, but almost implemented |
| `NullaryTypeClasses` | Deprecated |
| `NumDecimals` | Supported |
| `NumericUnderscores` | Supported |
| `OverlappingInstances` | Supported |
| `OverloadedLabels` | Unsupported, but easy to implement |
| `OverloadedLists` | Unsupported, but easy to implement |
| `OverloadedStrings` | Unsupported, but easy to implement |
| `PackageImports` | Supported |

| ParallelListComp | Unsupported |
|---|---|
| PartialTypeSignatures | Supported |
| PatternGuards | Supported |
| PatternSynonyms | Unsupported |
| PolyKinds | Unsupported |
| PostfixOperators | Supported |
| QuantifiedConstraints | Unsupported |
| QuasiQuotes | Unsupported |
| Rank2Types | Unsupported |
| RankNTypes | Unsupported |
| RebindableSyntax | Unsupported, but easy to implement |
| RecordWildCards | Supported |
| RecursiveDo | Unsupported |
| RoleAnnotations | Supported |
| Safe | Unsupported |
| ScopedTypeVariables | Unsupported |
| StandaloneDeriving | Supported |
| StarIsType | Supported |
| StaticPointers | Unsupported |
| Strict | Unsupported |
| StrictData | Unsupported |
| TemplateHaskell | Unsupported |
| TemplateHaskellQuotes | Unsupported |
| TraditionalRecordSyntax | Supported |
| TransformListComp | Unsupported |
| Trustworthy | Unsupported |
| TupleSections | Supported |
| TypeApplications | Supported |
| TypeFamilies | Unsupported |
| TypeFamilyDependencies | Unsupported |
| TypeInType | Unsupported |
| TypeOperators | Supported |
| TypeSynonymInstances | Supported |
| UnboxedSums | Unsupported |
| UnboxedTuples | Unsupported |
| UndecidableInstances | Supported |
| UndecidableSuperClasses | Supported |
| UnicodeSyntax | Supported |
| Unsafe | Unsupported |
| ViewPatterns | Unsupported |