Embedding Functional Logic Programming in Haskell via a Compiler Plugin

Kai-Oliver Prott[1][0000−0002−5795−6308], Finn Teegen[1][0000−0002−7905−3804], and Jan Christiansen[2][0000−0003−4911−8459]

[1] Kiel University, Kiel, Germany {kpr,fte}@uni-kiel.de
[2] Flensburg University of Applied Sciences, Flensburg, Germany
jan.christiansen@hs-flensburg.de

Abstract. We present a technique to embed a functional logic language in Haskell using a GHC plugin. Our approach is based on a monadic lifting that models the functional logic semantics explicitly. Using a GHC plugin, we get many language extensions that GHC provides for free in the embedded language. As a result, we obtain a seamless embedding of a functional logic language, without having to implement a full compiler. We briefly show that our approach can be used to embed other domain-specific languages as well. Furthermore, we can use such a plugin to build a full blown compiler for our language.

Keywords: functional programming · logic programming · DSL · Haskell · GHC plugin · monadic transformation

1 Introduction

Writing a compiler for a programming language is an elaborate task. For example, we have to write a parser, name resolution, and a type checker. In contrast, embedding a domain-specific language (DSL) within an already existing host language is much easier. If the host language is a functional programming language, such an embedding often comes with the downside of monadic overhead. To get the best of both worlds, we use a compiler plugin to embed a domain-specific language that does not require the user to write their code in a monadic syntax. Concretely, we implement a plugin for the Glasgow Haskell Compiler (GHC) that allows for a seamless embedding of functional logic programming in Haskell.[3] An additional benefit of our approach is that our embedded language supports a lot of Haskell's language extensions without additional effort.

Our lazy functional logic language is loosely based on the programming language Curry [19, 2]. However, instead of Curry's syntax, we have to restrict ourselves to Haskell's syntax. This is because at the moment[4] a GHC plugin cannot alter the static syntax accepted by GHC. Moreover, overlapping patterns in our language use Haskell's semantics of choosing the first rule instead of

[3] Available at https://github.com/cau-placc/ghc-language-plugin
[4] We plan to change this; see GHC issue 22401 [30].

introducing non-determinism as Curry does. In return we save a significant amount of work with our approach compared to implementing a full compiler for Curry. One of the compilers for the Curry programming language, the KiCS2 [7], translates Curry into Haskell. While the KiCS2 has around 120,000 lines of Haskell code, by using a GHC plugin we get the most basic functionality with approximately 7,500 lines of code. We have to be careful with these figures because we do not provide the same feature set. For example, KiCS2 provides free variables and unification whereas our plugin instead provides multi-parameter type classes. However, by adapting the underlying implementation, we could provide the remaining features of KiCS2 as well. Besides reducing the amount of code, when using a plugin we do not have to implement features that are already implemented in GHC from scratch. As examples, it took two master's theses to add multi-parameter type (constructor) classes to the KiCS2 [34, 25] while we get them for free.

The following example demonstrates the basic features of our approach using the Curry-like language. It enumerates permutations, the "hello world" example of the Curry programming language community. Here and in the following we label code blocks as follows: `Source` for code that is written in the embedded language, `Target` for corresponding Haskell code, which is generated by our plugin, and `Plugin` for code that is provided by us as part of our plugin.

Source

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

insert :: a → [a] → [a]
insert e []       = [e]
insert e (y : ys) = (e : y : ys) ? (y : insert e ys)

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

The first line activates our plugin. The function *insert* uses the (?) operator for introducing non-determinism to insert a given element at an arbitrary position in a list. That is, *insert* either adds e to the front of the list or puts y in front of the result of the recursive call. Building upon this, *permutations* non-deterministically computes a permutation of its input list by inserting the head of the list anywhere into a permutation of the remaining list.

Having the ability to write functional logic code in direct-style, i.e., without explicit monadic abstractions, is a great advantage. The code in direct-style is more concise as well as more readable, and makes it easy to compose non-deterministic and deterministic functions. To enable writing code in direct-style, our plugin transforms the code from direct-style to an explicit monadic representation behind the scenes. This transformation is type-safe and therefore allows for a seamless embedding of a functional logic language within Haskell.

To use the non-deterministic *permutations* operation in a Haskell program, we need to capture the non-determinism and convert the function to work on ordinary lists.

ghci> *eval1 permutations* $[1,2,3]$
$[[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]$

Here, *eval1* is essentially a wrapper that converts the 1-ary function *permutations* into a Haskell function that returns a list of all results and without non-determinism nested in data structures.[5] The result is a list of all permutations of the given input list.

We present the following contributions.

- We show how a GHC plugin can be used to embed a functional logic language in Haskell. By embedding the language via a plugin (Section 2, Section 3), we obtain the ability to write functional logic definitions, and can use the code written in the embedded language in Haskell.
- Besides saving the implementation of a parser and a type checker, we get additional language features like type classes, do-notation, bang patterns and functional dependencies for the functional logic language for free (Section 4).
- Although we demonstrate the embedding of a functional logic language in this paper, we argue that our approach can be generalized to languages other domain-specific languages as well (Section 5).

2 From Direct-Style to Explicit Monadic Style

Recall from Section 1 the implementation of a *permutations* functions in direct-style. Since Haskell does not natively support non-determinism, we need a way to represent our embedded programs in Haskell. To do this, we transform direct-style code to its explicit monadic counterpart. In this section, we will show the transformation rules from direct-style to monadic style.

2.1 A Lazy Functional-Logic Monad

Before we go into detail with the transformation, we need to introduce our functional logic language and its monadic implementation.

As seen before, our language provides non-deterministic branching. Since Haskell uses call-by-need evaluation, we want our language to be non-strict as well. In the context of functional logic programming, we also have to decide between using call-time choice or run-time choice [22]. Consider the function $double\ x = x + x$ applied to $1 ? 2$. With run-time choice, we would get the results 2, 3, and 4. The choice for the non-deterministic value in the argument of *double* is made independently for each ocurrence of x. However, obtaining the odd number 3 as a result of *double* is unintuitive. With call-time choice the choice for x is shared between all ocurrences. Thus, call-time choice only produces the result 2 and 4. For our language we settled for call-time choice since it is more intuitive in many cases [2].

[5] Using Template Haskell, we also provide an arity-independent wrapper function where one can specify a search strategy as well.

To model a functional logic language with call-by-need semantics and call-time choice, we use a technique presented by Fischer et al.[16]. In their paper, Fischer et al. use an operator *share* to turn a monadic computation into a computation that is evaluated at most once and is, thus, lazy. The *share* operator is part of the type class *MonadShare* that is defined as follows.[6]

```
Plugin
class Monad m ⇒ MonadShare m where
  share :: Shareable m a ⇒ m a → m (m a)
class Shareable m a where
  shareArgs :: a → m a
```

Intuitively, *share* registers its argument as a thunk in some kind of state and returns a new computation as a replacement for the shared expression. Whenever the returned computation is executed, it looks up if a value has been computed for the thunk. If this is the case, the said value is returned. Otherwise, the thunk is executed and the result value is saved for the future.

The additional type class *Shareable* needs to be defined for all monadically transformed data types. The class method *shareArgs* applies *share* recursively to the components of the transformed data types. Because data constructors can contain non-deterministic operations in their arguments, constructors take computations as arguments as well. This approach is well-known from modeling the non-strictness of Haskell in an actually pure programming language like Agda [1]. The usefulness of interleaving data types with computational effects has also been observed in more general settings [3].

Our lazy non-determinism monad *ND* is implemented using techniques from Fischer et al.[16]. The implementation provides the following two operations to introduce non-deterministic choices and failures by using the *Alternative* instance of our monad.

```
Source
(?) :: a → a → a
failed :: a
```

We omit the full implementation from the paper, because one could use any monadic implementation for a lazy functional logic language. In the future, we want to use the implementation from [20] for our language. However, the focus of this work is the embedding of the language, not the encoding of it.

2.2 The Transformation

Now we will show our transformation rules to compile direct-style to monadic style. In the context of a GHC plugin, we have used a similar transformation of Haskell code [35] to generate inverses of Haskell functions. A notable difference is that we include the modification with *share* to model call-by-need instead of just call-by-name.

We will use the *ND* type constructor in our transformation, although it works with any monadic type constructor that provides an implementation of *share*.

[6] The version presented in [16] has a more general type.

To increase readability of monadically transformed function types, we will use the following type.

```
newtype a →ND b = Func (ND a → ND b)                                    Plugin
```

2.3 Types

Our transformation on types replaces all type constructors with their non-deterministic counterparts. By replacing the type constructor $\to$ with $\to_{ND}$, the result and arguments of each functions are effectively wrapped in *ND* as well. Any quantifiers and constraints of a transformed type remain at the beginning of the type signature to avoid impredicativity. Additionally, we also wrap the outer type of a function in *ND* to allow an operation to introduce non-determinism before applying any arguments. In order to keep the original definitions of types available for interfacing the plugin with Haskell (Section 3.5), it is necessary to rename (type) constructors. This way we avoid name clashes by altering names of identifiers. Figure 1 presents the transformation of types, including the renaming of type constructors. An example for our transformation on types is given below.[7]

$$\llbracket \forall \alpha_1\ \ldots\ \alpha_n.\varphi \Rightarrow \tau \rrbracket^t \coloneqq \forall \alpha_1\ \ldots\ \alpha_n.\llbracket \varphi \rrbracket^t \Rightarrow \mathit{ND}\ \llbracket \tau \rrbracket^i \qquad \text{(Polymorphic type)}$$

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

$$\begin{aligned}
\llbracket \tau_1 \to \tau_2 \rrbracket^i &\coloneqq \llbracket \tau_1 \rrbracket^i \to_{ND} \llbracket \tau_2 \rrbracket^i && \text{(Function type)} \\
\llbracket \tau_1\ \tau_2 \rrbracket^i &\coloneqq \llbracket \tau_1 \rrbracket^i\ \llbracket \tau_2 \rrbracket^i && \text{(Type application)} \\
\llbracket \chi \rrbracket^i &\coloneqq \mathrm{rename}(\chi) && \text{(Type constructor)} \\
\llbracket \alpha \rrbracket^i &\coloneqq \alpha && \text{(Type variable)}
\end{aligned}$$

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

```
double :: Int → Int          Source          double :: ND (Int →ND Int)          Target
```

2.4 Data Type Declarations

To model non-strictness of data constructors, we need to modify most data type definitions and lift every constructor. As the (partial or full) application of a constructor can never introduce any non-determinism by itself, we neither have to transform the result type of the constructor nor wrap the function arrow. This allows us to only transform the parameters of constructors, because they are

[7] The transformed version of *Int* is called *Int* as well.

the only potential sources of non-determinism in a data type (Figure 2). The following code shows how a language plugin transforms the list data type. To improve readability we use regular algebraic data type syntax for the list type and its constructors.

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

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

$$
\begin{aligned}
&\textbf{data}\ \mathit{List}\ a = \mathit{Nil} \mid \mathit{Cons}\ a\ (\mathit{List}\ a) && \texttt{Source} \\
&\textbf{data}\ \mathit{List}_{ND}\ a = \mathit{Nil}_{ND} \mid \mathit{Cons}_{ND}\ (\mathit{ND}\ a)\ (\mathit{ND}\ (\mathit{List}_{ND}\ a)) && \texttt{Target}
\end{aligned}
$$

2.5 Functions

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

1. Each function arrow is wrapped in ND. Therefore, function definitions are replaced by constants that use multiple unary lambda expressions to introduce the arguments of the original function. All lambda expressions are wrapped in a *return* because function arrows are wrapped in a ND type.
2. We have to extract a value from the monad using $(\gg\!=)$ before we can pattern match on it. As an implication, we cannot pattern match directly on the argument of a lambda or use nested pattern matching, as arguments of a lambda and nested values are potentially non-deterministic and have to be extracted using $(\gg\!=)$ again.
3. Before applying a function to an argument, we first have to extract the function from the monad using $(\gg\!=)$. As each function arrow is wrapped separately, we extract each parameter of the original function.

Implementing this kind of transformation in one pass over a complex Haskell program is challenging. Thus, we simplify each program first. As the most difficulties arise from pattern matching, we perform pattern match compilation to get rid of complex and nested patterns. In contrast to the transformation done by GHC, we want a Haskell AST instead of Core code as output. Thus, we cannot re-use GHC's existing implementation and use a custom transformation that is similar to the one presented in [39]. Note that we still preserve any non-exhaustiveness warnings and similar for pattern matching by applying GHC's

pattern match checker before our transformation. For the remainder of this subsection we assume that our program is already desugared, that is, all pattern matching is performed with non-nested patterns in case expressions.

$$
\begin{aligned}
[\![v]\!] &\coloneqq v && \text{(Variable)}\\
[\![\lambda x \to e]\!] &\coloneqq \mathit{return}\ (\mathit{Func}\ (\lambda y \to \mathrm{alias}(y, x, [\![e]\!]))) && \text{(Abstraction)}\\
[\![e_1\ e_2]\!] &\coloneqq [\![e_1]\!] \gg\!= (\lambda(\mathit{Func}\ f) \to f\ [\![e_2]\!]) && \text{(Application)}\\
[\![C]\!] &\coloneqq \mathit{return}\ (\mathit{Func}\ (\lambda y_1 \to \ldots\\
&\qquad \mathit{return}\ (\mathit{Func}\ (\lambda y_n \to \ldots\\
&\qquad \mathit{return}\ (\mathrm{rename}(C)\ y_1\ \ldots\ y_n))))) && \text{(Constructor)}\\
[\![\ \mathbf{case}\ e\ \mathbf{of}\ \{br_1; \ldots; br_n\}]\!] &\coloneqq [\![e]\!] \gg\!= (\lambda y \to \mathbf{case}\ y\ \mathbf{of}\ \{[\![br_1]\!]^b; \ldots; [\![br_n]\!]^b\}) && \text{(Case Expression)}\\
[\![C\ x_1\ \ldots\ x_n \to e]\!]^b &\coloneqq \mathrm{rename}(C)\ y_1\ \ldots\ y_n \to\\
&\qquad \mathrm{alias}(y_1, x_1, \ldots\ \mathrm{alias}(y_n, x_n, [\![e]\!])) && \text{(Case Branch)}\\
\mathrm{alias}(v_{new}, v_{old}, e) &\coloneqq \begin{cases} \mathit{share}\ v_{new} \gg\!= \lambda v_{old} \to e & \text{if } v_{old} \text{ occurs at least twice in } e\\ \mathbf{let}\ v_{old} = v_{new}\ \mathbf{in}\ e & \text{otherwise} \end{cases} && \text{(Aliasing)}
\end{aligned}
$$

Figure 3: Expression lifting $[\![\cdot]\!]$ (y and y_1 to y_n are fresh variables)

Figure 3 presents the rules of our transformation. Note that any bindings introduced by a lambda or case expression are shared using the alias rule. We have omitted rules for let declarations, since recursive bindings are a bit tricky to get right. They are supported by our implementation as long as the monadic type has a sensible *MonadFix* instance. This is the case for our simple implementation.

To give an example of our whole lifting, the *permutations* example from the introduction with minor simplifications looks as follows.

```
                                                                          Target
insertND :: ND (a →ND ListND a →ND ListND a)
insertND = return (Func (λarg1 → share arg1 >>= λe →
             return (Func (λarg2 → share arg2 >>= λxs → xs >>= λcase
  NilND          → return (ConsND e (return NilND))
  ConsND y ys → return (ConsND e (return (ConsND y ys))) <|>
                  return (ConsND y (insertND >>= λ(Func f1) → f1 e
                                             >>= λ(Func f2) → f2 ys))))))
permutationsND :: ND (ListND a →ND ListND a)
permutationsND = return (Func (λarg → share arg >>= λarg' → arg' >>= λcase
  NilND          → return NilND
  ConsND x xs → insertND >>= λ(Func f1) → f1 x >>= λ(Func f2) →
                  f2 (permutationsND >>= λ(Func f3) → f3 xs)))
```

The transformation made the code significantly more complex, where most of the complexity arises from the modeling of laziness. This also emphasizes that writing code in direct-style is more readable for lazy fuctional logic programming.

3 Plugin Core

In this section, we motivate the use of a plugin and discuss its core. In particular we present how the transformation to monadic code is achieved.

3.1 Why a GHC Plugin?

The GHC allows extending its functionality using a compiler plugin API. Using this API, we can change the behavior of the type check and analyze or transform code at various intermediate stages.

There are some alternatives to using a plugin, for example, metaprogramming via Template Haskell [32]. One crucial point is that our transformation (e.g. in Section 3.4) requires type information that is not present when using Template Haskell. Even with typed Template Haskell, we do not get all type annotations that we need for the transformation. Additionally, we aim to provide good quality error messages produced by GHC itself (Section 3.3), for which we have no idea how to properly achieve this with a transformation using (typed) Template Haskell. We could do the transformation on the level of GHC's intermediate language Core, which would eliminate the need for our own pattern match transformation and reduce the amount of syntax constructs we have to take into account. But a short evaluation showed that some aspects of the transformation are harder to achieve on core level, for example, generating additional instances, modifying data types, and adding additional type constraints to functions.

3.2 Plugin Phases

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. The "Import Check" and the "Import Constraint Solver" are both concerned with the correct handling of imports. The former checks that each imported module has been compiled with the same language plugin, while the latter resolves all type mismatches that arise from the transformation (Section 3.3). At last, the main work is done in the "Transformation" phase, which is again divided into four sub-phases:

1. Transforming data types to model non-strictness of data constructors. Type classes are also transformed in this sub-phase (not shown in this paper).
2. Compiling and simplifying pattern matching to make the implementation of sub-phase 4 simpler.
3. Deriving instances of internal type classes for all data types, because our transformation requires each data type to have certain instances (like *Shareable* to model sharing of non-deterministic computations).

4. Transforming function definitions to achieve our desired semantics. Here we use the rules shown in Section 2.2

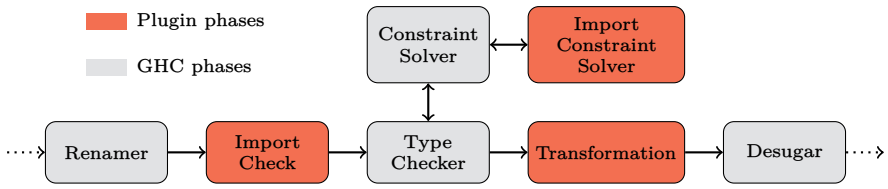

Figure 4: Extension points used by our plugin

3.3 Handling of Imported Definitions

We consider the following constant where *double* is imported from a different module.

Source

$$example :: Int$$
$$example = double\ (23 :: Int)$$

If the definition was not imported from a module that has already been compiled by the plugin, it will not be compatible with the current module. Therefore we check that each module only imports plugin-compiled modules and abort compilation with an error message otherwise. A programmer can explicitly mark a non-plugin module as safe to be imported, but its the programmers obligation to ensure that all functions in that module have plugin-compatible types. Let us assume the import of *double* passes the "Import Check". Before our transformation takes place, GHC type checks the whole module. The definition of *example* will cause a type error, because the non-determinism is explicit in the transformed variant $double :: ND\ (Int \to_{ND} Int)$ that has already been compiled by the plugin. In fact, any imported function (and data type) has a lifted type and, thus, using such a function will be rejected by GHC's type check. However, the transformed version of *example* will later be type correct and we thus want to accept the definition.

To overcome this issue we use a constraint solver plugin. When GHC type checks *example*, it realizes that *double* should have type $Int \to Int$ from looking at the argument $23 :: Int$ and the result type of *example*. However, *double* was imported and its type is known to be $ND\ (Int \to_{ND} Int)$. The type checker will now create an equality constraint[8]

$$ND\ (Int \to_{ND} Int) \sim (Int \to Int)$$

[8] The operator $(\sim)$ denotes homogeneous (both kinds are the same) equality of types.

and sends it to GHC's constraint solver to (dis-)prove. Naturally, GHC cannot make any progress proving this equality and instead forwards it to our constraint solver plugin. There we un-transform the type, that is, we remove the outer *ND* and un-transform the transformed function arrow ($\rightarrow_{ND}$) to obtain our new left-hand side for the equality constraint:

$$(Int \rightarrow Int) \sim (Int \rightarrow Int).$$

We pass this new constraint back to GHC, where it might be solved to show that the untransformed program is well-typed. That is, we consider a type τ_1 and its non-deterministic variant τ_2 to be equal by using the following equation.

$$ND\ \tau_2 \sim \tau_1$$

This approach is similar to the "type consistency" relation in gradual typing [33]. For details about GHC's constraint solver, we refer the interested reader to [36].

Besides making transformed and untransformed types compatible the constraint solver plugin also needs to automatically solve any constraints that mention the *Shareable* type class. GHC cannot always discharge these constraints before our main transformation has taken place.

Type Errors Our plugin ensures that ill-typed programs are not accepted and returns a meaningful type error. Consider the following ill-typed program.

```
typeError :: Int                                                    Source
typeError = double True
```

Using our plugin, GHC generates the following type error as if *double* actually had the untransformed type.

```
– Couldn't match expected type Int with actual type Bool
– In the first argument of double, namely True
  In the expression: double True
  In an equation for typeError: typeError = double True
```

In fact, any error messages produced by GHC for modules with our plugin activated are (mostly) the same as if the user would have written the identical code in "vanilla" Haskell. There are only a few deliberate exceptions where we augment certain errors or warnings with more information. We achieve these error messages by transforming the code after the type check and by deliberately un-transforming the types of imported functions.

3.4 Sharing Polymorphic Expressions

We have already introduced the *share* operator at the beginning of the section. For call-by-need evaluation, this operator is implemented as presented in [16] and requires a type class constraint *Shareable* $(m :: Type \rightarrow Type)$ $(a :: Type)$. The constraint enforces that values of type a can be shared in the monad m.

Instances of this type class are automatically derived by the plugin for every user-defined data type via generic deriving [26].

Ensuring that all types in a type signature are *Shareable* is easy to achieve if the type signature is monomorphic or only contains type variables with the simple kind *Type*. For variables with other kinds, e.g., $f :: Type \to Type$, we cannot add a constraint for that variable as $Shareable\ m\ f$ is ill-kinded. So, what can one do if Haskell's type and class system is not powerful enough? Just enable even more language extensions![9] In our case, we can solve the problem by using the extension *QuantifiedConstraints* [6]. What we really want to demand for f in the example above is that, for every type x that is *Shareable*, the type $f\ x$ should be *Shareable* as well. We can express exactly this requirement by using a $\forall$ quantifier in our constraint. This is demonstrated by the following example.

Source
```
void :: Functor f ⇒ f a → f ()
```

Target
```
void :: (Functor f, Shareable ND a
        ,∀x.Shareable ND x ⇒ Shareable ND (f x))
      ⇒ ND (f a →ND f ())
```

We can extend this scheme to even more complex kinds, as long as it only consists of the kind of ordinary Haskell values (*Type*) and the arrow kind constructor ($\to$). Language extensions that allow the usage of other kinds are not supported at the moment. To add *Shareable* constraints in types, we replace the first rule in Figure 1 by the rules in Figure 5.

$$[\![\forall\alpha_1\ \ldots\ \alpha_n.\varphi \Rightarrow \tau]\!]^t \coloneqq \forall\alpha_1\ \ldots\ \alpha_n.([\![\varphi]\!]^t \cup \langle [\![\alpha_1]\!]^s, \ldots, [\![\alpha_n]\!]^s\rangle) \Rightarrow [\![\tau]\!]^t \qquad \text{(Polymorphic type)}$$

$$[\![v :: (\kappa_1 \to \ldots \to \kappa_n)]\!]^s \coloneqq \forall(\beta_1 :: \kappa_1)\ \ldots\ (\beta_{n-1} :: \kappa_{n-1}).\langle [\![\beta_1]\!]^s, \ldots, [\![\beta_{n-1}]\!]^s\rangle \Rightarrow Shareable\ ND\ (v\ \beta_1\ \ldots\ \beta_{n-1}) \qquad \text{(Type variable of kind } \kappa_1 \to \ldots \to \kappa_n)$$

$$[\![v :: Type]\!]^s \coloneqq Shareable\ ND\ v \qquad \text{(Type variable of kind } Type)$$

Figure 5: Type lifting $[\![\cdot]\!]^t$ with *Shareable* constraints ($\cup$ joins two contexts, $\beta_1\ \ldots\ \beta_{n-1}$ are fresh variables)

3.5 Interfacing with Haskell

At the end of the day, we want to call the functions written in our embedded language from within Haskell and extract the non-deterministic results in some kind of data structure. Because data types are transformed to support non-strict

[9] Use this advice at your own risk.

data constructors, we do not only have to encapsulate non-determinism at the top-level of a result type, but effects that are nested inside of data structures as well. To trigger all non-determinism inside a data structure, we convert data types to some kind of normal form, where only top-level non-determinism is possible. We also translate data types defined in our embedded language into their original counterparts. This way, we can embed our new language within Haskell as a DSL.

Consider the following example with a list of n non-deterministic coins that are either *False* or *True*.

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

manyCoins :: Int → [Bool]
manyCoins n = [coin | _ ← [1 .. n]]
```

The transformed type will be $ND\ (Int \to_{ND} List_{ND}\ Bool_{ND})$. To use the result of *manyCoins* in plain Haskell, it is convenient to convert the list with non-deterministic components into an ordinary Haskell list. Therefore, we move any non-determinism occurring in the elements of the list to the outermost level.

The conversion to normal form is automatically derived for all data types by generating instances of the type class *NormalForm*.[10]

Plugin
```
class NormalForm a b where
  nf :: a → ND b
  embed :: b → a
```

Here, the type variable a will be a transformed type, while b will be the corresponding original definition. The function *nf* converts a value with nested non-determinism into its normal form. To convert a data type to normal form, we convert the arguments to normal form and chain the values together with $(\gg\!=)$. The result of *nf* contains non-determinism only at the outermost level and the data type is transformed back to its original definition.

We also support conversion in the other direction. The function *embed* converts an untransformed data type to its transformed representation by recursively embedding all constructor arguments. Note that this type class is not required for the monadic transformation itself, but for the interface between Haskell and our embedded language.

For the list data type and its non-deterministic variant $List_{ND}$ from Section 2.4, we define the *NormalForm* instance as follows.

Target
```
instance NormalForm a b ⇒ NormalForm (List_ND a) (List b) where
  nf Nil_ND         = return Nil
  nf (Cons_ND x xs) = liftM2 Cons (x >>= nf) (xs >>= nf)
  embed Nil         = Nil_ND
  embed (Cons x xs) = Cons_ND (return (embed x)) (return (embed xs))
```

[10] It is similar to the type class *Convertible* from [16]

3.6 Compatibility

The plugin makes heavy use of GHC's internal APIs and is thus only compatible with a specific GHC version. Development started with GHC 8.10, but some of the experiments we conducted with higher-rank types required us to enable the "Quick Look" approach for impredicative types [31] introduced in GHC 9.2. The difficulty of upgrading the plugin to support a new GHC version depends on the complexity of GHC's internal changes, but even the upgrade from 8.10 to 9.2 did not take long although there were significant changes to GHC under the hood.

Also note that new GHC language features are not supported automatically, even if we upgrade the plugin to the new GHC version. For example, consider Linear Haskell as introduced in GHC 9.0. With this version, GHC's internal representation for function types changed to accommodate linearity information. When we upgraded the plugin past this major version, we had to adapt to the new representation. However, without careful considerations during the transformation, the linearity information is lost or not accurately reflected in the final code. Thus, without explicitly treating the linearity information correctly (what that means exactly is still left to determine), the plugin does not support Linear Haskell properly. But when linear types are not used, the transformation succeeds for the new GHC version.

4 Inherited Features

In this section we discuss GHC features that our plugin supports for the corresponding embedded language as well. Some of these features are supported out-of-the-box, that is, just by using a GHC plugin to implement a language. A few type-level and type class extensions are among these features, for instance. Other features required us to make small adaptions to the plugin, for example, to add additional cases to the transformation.

To demonstrate the benefits of our approach, we revisit the example from Section 1. We consider a variant that uses a multi-parameter type class with a functional dependency [24] to abstract from list-like data structures and view patterns [37, 14] to define a generalized version of *permutations*. We cannot write this program using any of the available Curry compilers as none of them support the required language features. That is, using a plugin we can not only implement a non-deterministic language in 7,500 lines of code, we also get features for free that no other full-blown Curry compiler with more than 120,000 lines of code supports.

Source

```
class ListLike l e | l → e where
  nil :: l
  cons :: e → l → l
  uncons :: l → Maybe (e, l)
permutations :: ListLike l e ⇒ l → l
permutations (uncons → Nothing)     = nil
permutations (uncons → Just (x, xs)) = ins x (permutations xs)
```

$$\begin{aligned}\textbf{where } ins\ e\ (uncons \to Nothing) &= cons\ e\ nil \\ ins\ e\ (uncons \to Just\ (x, xs)) &= cons\ e\ (cons\ x\ xs)\ ?\ cons\ x\ (ins\ e\ xs)\end{aligned}$$

Ad-hoc polymorphism via type classes is one of Haskell's most useful features for writing concise programs. Properly supporting type classes as well as extensions surrounding them like multi-parameter type classes, functional dependencies and flexible contexts is straightforward. We can just rely on GHC for solving type class constraints and for implementing type classes via dictionary transformation [23, 38]. Apart from minor technical differences, we can handle functions from a type class instance like any other function during our monadic transformation. As type classes are a complex feature, we were kind of surprised that a lot of Haskell's type class extensions are easy to support.

Most language extensions required little to no additional lines of code to be supported by our plugin. Some of them are among the most commonly used extensions on Hackage [10, 9]. From the 30 most used extensions, 19 are currently supported by our plugin. Figure 6 lists a selection of these extensions and their respective rank in the list of most commonly used GHC extensions. The extensions in the column labeled "Supported out-of-the-box" are orthogonal to the plugin transformation and are supported without any additional effort. The column labeled "Small adaptions" in Figure 6 lists extensions that are supported after minor changes. Most of these extensions add syntactic sugar that is simple enough to transform.

Figure 6: Some extensions with their rank in most commonly used GHC extensions

Supported out-of-the-box		Small adaptions
2. `FlexibleInstances`	5. `MultiParamTypeClasses`	1. `OverloadedStrings`
3. `FlexibleContexts`	13. `RecordWildCards`	15. `LambdaCase`
4. `ScopedTypeVariables`	14. `TypeOperators`	20. `TupleSections`

Unsupported Features and Limits Although our plugin can handle many language extensions, some of GHC's more feature-rich and expressive extensions remain unsupported. We can group the problems for each extension into three categories:

1. The extension enriches GHC's type system. Since our transformation works on the typed syntax tree, our transformation would have to consider more complex type expressions, which is challenging on a technical level to get right. This is the case for, e.g., type families and higher-rank types. For higher-rank types, one of the problems is that they lead to impredicativity in combination with our transformation.
2. The extension allows the definition of more flexible data types. Our plugin has to derive instances of some type classes (e.g., Section 3.5) automatically, which gets more challenging for complex data types as well. Examples for such an extension are existential data types.

3. The extension introduces a lot of syntactical sugar that requires a great deal of work to support. One prominent example for this category is the extension *RebindableSyntax*, which just introduces a lot of corner cases to consider during our monadic transformation. However, there is no reason for extensions in this category to be hard or impossible to support.

5 Embedding Other Languages

To show the applicability of our approach for different languages, we have embedded additional languages using a plugin: a strict functional language, a probabilistic language, and a language with dynamic information flow control. For brevity, we do not include these in the paper but provide them on GitHub.[11] All these additional languages must use Haskell's syntax too. However, we are working on lifting that restriction by extending the GHC plugin API with the required capabilities. Note that all plugins are built upon a generalized variant of the plugin we presented in this paper. Implementing the monads that model the effects of these languages is as simple or as complex as implementing a monad in Haskell – with the same type of errors and warnings, for example. Apart from the concrete effect implementation, almost all of our plugin code is shared between the three examples. The implementations of these languages are approximately 400 lines of code each.

Note that the additional languages do not use call-by-need as their evaluation strategy. This is not a problem, however, as we generalized our monadic transformation to model call-by-value and call-by-name semantics as well. These languages also need different implementations for the *share* operator. Details on implementing *share* for different evaluation strategies can be found in [27].

A major goal for future research is to embed even more languages. We plan to enlarge the class of languages that can be embedded using our plugin. Currently, we can use all languages whose semantics can be modeled using a standard monad that are not parametrized over an additional parameter. For example, at the moment we cannot use our plugin to model a language with algebraic effects on the basis of a free monad. We also plan to investigate the generalization to more different forms of monads like graded or indexed monads.

6 Related Work

There are three different groups of related work.

6.1 GHC Plugins

The first group of related work uses GHC plugins. The GHC plugin infrastructure has been used to extend Haskell with different kinds of features. Using constraint solver plugins, Gundry [18] adds support for units of measure and Diatchki

[11] See README file in the linked repository from Footnote 3.

[12] adds type-level functions on natural numbers. Di Napoli et al. [11] use a combination of plugins to extend Haskell with liquid types and Breitner [8] uses a combination of Template Haskell and a Core plugin to provide inspection testing. We also use a combination of plugins, we use a constraint solver plugin to recover from type errors and a source plugin [29] that uses the renamer and the type checker extension point for the transformation. However, none of the other plugins transform the compiled code to such an extent as our plugin since they do not aim for a thorough transformation. The only plugin that is similar is one that compiles GHC Core code to abstractions from category theory [13]. The approach is suitable for the definition of DSLs, but only supports monomorphic functions and values. In contrast, our plugin aims to model a full functional programming language with implicit effects.

6.2 Monadic Intermediate Languages

Our transformation basically models the denotational semantics of a language explicitly. Peyton Jones et al. [28] have applied a similar approach to model Haskell and ML in a single monadic intermediate language. However, although the underlying idea bears some resemblance, the contributions of our work are different. While existing work focuses on common intermediate languages, we use Haskell and GHC for more than just the compiler backend.

Transforming an impure functional language language into monadic code has been applied to model Haskell in various proof assistants. For example, a Core plugin for GHC is used by [1] to generate Agda code that captures Haskell's semantics via an explicit monadic effect.

6.3 Embedding Languages

Last but not least in the Scheme and Racket family of programming languages users can define new languages by using advanced macro systems [17]. For example, there exists a lazy variant of PLT Scheme for teaching [4] and an implementation of miniKanren, a logic programming language [21]. Felleisen et al. [15] even aim for language-oriented programming that facilitates the creation of DSLs, the development in these languages and the integration of multiple of these languages. We do not aim for language-oriented programming but to simplify writing and embedding research languages within a different ecosystem, namely the GHC with its statically typed, purely functional programming language. We target all those research prototype languages that could as well be implemented as a monadic variant of Haskell without the syntactic monadic overhead.

7 Conclusion

In this paper, we have shown how to embed a functional logic language into Haskell. Using a compiler plugin allows us to use direct-style syntax for the embedded language while retaining good quality error messages. In the future,

we want to extend the approach as mentioned in Section 5 and add support for more of GHC's language extensions. For example, we plan to investigate the support of more current features like linear types [5] as well as extensions like higher-rank types.

Acknowledgements We thank our anonymous reviewers and Michael Hanus for their helpful comments and fruitful discussions.

Bibliography

[1] Abel, A., Benke, M., Bove, A., Hughes, J., Norell, U.: Verifying Haskell programs using constructive type theory. In: Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell. pp. 62–73. ACM Press, New York, NY, USA (2005). https://doi.org/10.1145/1088348.1088355

[2] Antoy, S., Hanus, M.: Functional logic programming. Communications of the ACM **53**(4), 74 (2010). https://doi.org/10.1145/1721654.1721675

[3] Atkey, R., Johann, P.: Interleaving data and effects. Journal of Functional Programming **25** (2015). https://doi.org/10.1017/S0956796815000209

[4] Barzilay, E., Clements, J.: Laziness without all the hard work: Combining lazy and strict languages for teaching. In: Proceedings of the 2005 Workshop on Functional and Declaritive Programming in Education - FDPE '05. p. 9. ACM Press, Tallinn, Estonia (2005). https://doi.org/10.1145/1085114.1085118

[5] Bernardy, J.P., Boespflug, M., Newton, R.R., Jones, S.P., Spiwack, A.: Linear Haskell: Practical linearity in a higher-order polymorphic language. Proceedings of the ACM on Programming Languages **2**(POPL), 1–29 (Jan 2018). https://doi.org/10.1145/3158093

[6] Bottu, G.J., Karachalias, G., Schrijvers, T., Oliveira, B.C.d.S., Wadler, P.: Quantified class constraints. In: Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell. pp. 148–161. Haskell 2017, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3122955.3122967

[7] Braßel, B., Hanus, M., Peemöller, B., Reck, F.: KiCS2: A new compiler from Curry to Haskell. In: International Workshop on Functional and Constraint Logic Programming. pp. 1–18. Springer Berlin Heidelberg (2011). https://doi.org/10.1007/978-3-642-22531-4_1

[8] Breitner, J.: A promise checked is a promise kept: Inspection testing. In: Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell. pp. 14–25. Haskell 2018, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3242744.3242748

[9] Breitner, J.: GHC Extensions stats (Nov 2020), https://gist.github.com/nomeata/3d1a75f8ab8980f944fc8c845d6fb9a9

[10] Breitner, J.: [ghc-steering-committee] Prelimary GHC extensions stats (Nov 2020), https://mail.haskell.org/pipermail/ghc-steering-committee/2020-November/001876.html

[11] Di Napoli, A., Jhala, R., Löh, A., Vazou, N.: Liquid Haskell as a GHC Plugin - Haskell implementors workshop 2020 (Aug 2020), https://icfp20.sigplan.org/details/hiw-2020-papers/1/Liquid-Haskell-as-a-GHC-Plugin

[12] Diatchki, I.S.: Improving Haskell types with SMT. In: Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell. pp. 1–10. Haskell '15, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2804302.2804307

[13] Elliott, C.: Compiling to categories. Proceedings of the ACM on Programming Languages **1**(ICFP) (Aug 2017). https://doi.org/10.1145/3110271, https://doi.org/10.1145/3110271

[14] Erwig, M., Peyton Jones, S.: Pattern guards and transformational patterns. Electronic Notes in Theoretical Computer Science **41**(1), 3 (2001). https://doi.org/10.1016/S1571-0661(05)80540-7

[15] Felleisen, M., Findler, R.B., Flatt, M., Krishnamurthi, S., Barzilay, E., McCarthy, J., Tobin-Hochstadt, S.: A programmable programming language. Communications of the ACM **61**(3), 62–71 (Feb 2018). https://doi.org/10.1145/3127323

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

[17] Flatt, M.: Composable and compilable macros: You want it when? In: Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming. pp. 72–83. ICFP '02, Association for Computing Machinery, New York, NY, USA (2002). https://doi.org/10.1145/581478.581486

[18] Gundry, A.: A typechecker plugin for units of measure: Domain-specific constraint solving in GHC Haskell. In: Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell. pp. 11–22. Haskell '15, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2804302.2804305

[19] Hanus, M., Kuchen, H., Moreno-Navarro, J.: Curry: A Truly Functional Logic Language. In: Proc. ILPS'95 Workshop on Visions for the Future of Logic Programming. pp. 95–107 (Aug 1995)

[20] Hanus, M., Prott, K.O., Teegen, F.: A monadic implementation of functional logic programs. In: Proceedings of the 24th International Symposium on Principles and Practice of Declarative Programming. PPDP '22, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3551357.3551370

[21] Hemann, J., Friedman, D.P., Byrd, W.E., Might, M.: A small embedding of logic programming with a simple complete search. In: Proceedings of the 12th Symposium on Dynamic Languages. pp. 96–107. DLS 2016, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2989225.2989230

[22] Hussmann, H.: Nondeterministic algebraic specifications and nonconfluent term rewriting. Journal of Logic Programming **12**, 237–255 (1992). https://doi.org/10.1016/0743-1066(92)90026-Y

[23] Jones, M.P.: A system of constructor classes: Overloading and implicit higher-order polymorphism. In: Proceedings of the Conference on Functional Programming Languages and Computer Architecture - FPCA '93. pp. 52–61. ACM Press, Copenhagen, Denmark (1993). https://doi.org/10.1145/165180.165190

[24] Jones, M.P.: Type Classes with Functional Dependencies. In: Programming Languages and Systems. vol. 1782, pp. 230–244. Springer Berlin Heidelberg, Berlin, Heidelberg (2000). https://doi.org/10.1007/3-540-46425-5_15

[25] Krüger, L.E.: Extending Curry with multi parameter type classes (in german). Master's Thesis, Christian-Albrechts-Universität zu Kiel, Kiel, Germany (2021), https://www.informatik.uni-kiel.de/~mh/lehre/abschlussarbeiten/msc/Krueger_Leif_Erik.pdf

[26] Magalhães, J.P., Dijkstra, A., Jeuring, J., Löh, A.: A generic deriving mechanism for Haskell. In: Proceedings of the Third ACM Haskell Symposium on Haskell. pp. 37–48. Haskell '10, Association for Computing Machinery, New York, NY, USA (2010). https://doi.org/10.1145/1863523.1863529

[27] Petricek, T.: Evaluation strategies for monadic computations. In: Proceedings of Mathematically Structured Functional Programming. vol. 76, pp. 68–89 (Feb 2012). https://doi.org/10.4204/EPTCS.76.7

[28] Peyton Jones, S., Shields, M., Launchbury, J., Tolmach, A.: Bridging the gulf: A common intermediate language for ML and Haskell. In: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 49–61. POPL '98, Association for Computing Machinery, New York, NY, USA (1998). https://doi.org/10.1145/268946.268951

[29] Pickering, M., Wu, N., Németh, B.: Working with source plugins. In: Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell - Haskell 2019. pp. 85–97. ACM Press, Berlin, Germany (2019). https://doi.org/10.1145/3331545.3342599

[30] Prott, K.O.: Plugin-swappable parser (Nov 2022), https://gitlab.haskell.org/ghc/ghc/-/issues/22401

[31] Serrano, A., Hage, J., Peyton Jones, S., Vytiniotis, D.: A quick look at impredicativity. Proceedings of the ACM on Programming Languages **4**(ICFP), 1–29 (Aug 2020). https://doi.org/10.1145/3408971

[32] Sheard, T., Jones, S.P.: Template Meta-Programming for Haskell. In: Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell. pp. 1–16. Haskell 2002, Association for Computing Machinery, New York, NY, USA (2002). https://doi.org/10.1145/581690.581691

[33] Siek, J., Taha, W.: Gradual typing for functional languages. In: Proceedings of the 2006 Scheme and Functional Programming Workshop. pp. 81–92 (2006)

[34] Teegen, F.: Extending Curry with type classes and type constructor classes (in german). Master's Thesis, Christian-Albrechts-Universität zu Kiel, Kiel, Germany (2016), https://www.informatik.uni-kiel.de/~mh/lehre/abschlussarbeiten/msc/Teegen.pdf

[35] Teegen, F., Prott, K.O., Bunkenburg, N.: Haskell^{-1}: Automatic function inversion in Haskell. In: Proceedings of the 14th ACM SIGPLAN International Symposium on Haskell. pp. 41–55. Haskell 2021, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3471874.3472982

[36] Vytiniotis, D., Jones, S.P., Schrijvers, T., Sulzmann, M.: OutsideIn(X) Modular type inference with local assumptions. Journal of Functional Programming **21**(4-5), 333–412 (2011). https://doi.org/10.1017/S0956796811000098

[37] Wadler, P.: Views: A way for pattern matching to cohabit with data abstraction. In: Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages - POPL '87. pp. 307–313. ACM Press, Munich, West Germany (1987). https://doi.org/10.1145/41625.41653

[38] Wadler, P., Blott, S.: How to make ad-hoc polymorphism less ad hoc. In: Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL '89. pp. 60–76. ACM Press, Austin, Texas, United States (1989). https://doi.org/10.1145/75277.75283

[39] Wadler, P.: Efficient Compilation of Pattern-Matching. In: Jones, S.L.P. (ed.) The Implementation of Functional Programming Languages, pp. 78–103. Prentice-Hall (1987). https://doi.org/10.1016/0141-9331(87)90510-2