Haskell^{-1}

Automatic Function Inversion in Haskell

Finn Teegen
University of Kiel
Germany
fte@informatik.uni-kiel.de

Kai-Oliver Prott
University of Kiel
Germany
kpr@informatik.uni-kiel.de

Niels Bunkenburg
University of Kiel
Germany
nbu@informatik.uni-kiel.de

Abstract

We present an approach for automatic function inversion in Haskell. The inverse functions we generate are based on an extension of Haskell's computational model with non-determinism and free variables. We implement this functional logic extension of Haskell via a monadic lifting of functions and type declarations. Using inverse functions, we additionally show how Haskell's pattern matching can be augmented with support for *functional patterns*, which enable arbitrarily deep pattern matching in data structures. Finally, we provide a plugin for the Glasgow Haskell Compiler to seamlessly integrate inverses and functional patterns into the language, covering almost all of the Haskell2010 language standard.

CCS Concepts: • **Software and its engineering** → **Functional languages**; **Constraint and logic languages**; *Compilers*; Source code generation.

Keywords: Haskell, inversion, partial inversion, monadic transformation, pattern matching, GHC plugin

ACM Reference Format:
Finn Teegen, Kai-Oliver Prott, and Niels Bunkenburg. 2021. Haskell^{-1}: Automatic Function Inversion in Haskell. In *Proceedings of the 14th ACM SIGPLAN International Haskell Symposium (Haskell '21), August 26–27, 2021, Virtual, Republic of Korea.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3471874.3472982

1 Introduction

While functions compute some output for a given input, *inverse functions* should in turn compute the input for a given output. Invertible functions are an important concept: common applications include parser/printer [33, 46], compressor/decompressor [53], and serializer/deserializer [28]—all of which can be seen as pairs of functions with their inverses.

Function inversion [35] describes the task of generating inverse functions automatically. While automatic inversion does not necessarily always yield the most performant implementation, it is, for example, well suited to obtain reference implementations for automatic testing. Automatic inversion can also be used to synthesize parallel divide-and-conquer algorithms [37].

In reality, however, there are many functions which have no unique inverse. For instance, non-injective functions have multiple inputs that lead to the same output and, thus, unique inverses for such functions do not exist. In contrast, injective functions are known to have unique inverses that are in many cases efficient. For this reason, many other approaches for automatic function inversion focus on injective functions only [4, 16, 39]. However, this is often not practical in a general-purpose language like Haskell, where non-injective functions are quite common. As an example, consider the function (++) that concatenates two input lists.

```
(++) :: [a] -> [a] -> [a]
[]       ++ ys = ys
(x : xs) ++ ys = x : xs ++ ys
```

For any non-empty output list, there are at least two possible combinations of input lists. For example, the expressions [ ] ++ [1, 2], [1] ++ [2], and [1, 2] ++ [ ] all evaluate to [1, 2].

The problem of inverting non-injective functions can be tackled by allowing inverse functions to return multiple results, e.g., in a list structure. A possible type for the inverse of (++) could be the following one.

```
(++)^-1 :: [a] -> [([a], [a])]
```

A call to this inverse function then might look as follows.

```
ghci> (++)^-1 [1,2]
[([],[1,2]),([1],[2]),([1,2],[])]
```

Note that the type of inverse functions is not determined by the arity of the original function's equation(s), but by the arity of the original function's type. This way, it is possible to decide if a use of an inverse is well-typed based only on

Haskell '21, August 26–27, 2021, Virtual, Republic of Korea

ACM ISBN 978-1-4503-8615-9/21/08...$15.00
https://doi.org/10.1145/3471874.3472982

the original function's type, without reference to the original function's declaration.[1]

In logic programming languages, inverse computation is often achieved without much effort. For instance, in Prolog we can call predicates with free variables as arguments to find variable bindings such that the predicate is satisfied.

prolog> $append\,(Xs, Ys, [1,2])$.
$Xs = [\,]$, $Ys = [1,2]$;
$Xs = [1]$, $Ys = [2]$;
$Xs = [1,2]$, $Ys = [\,]$.

Our approach to implement function inversion in Haskell is similar to Prolog in the sense that we want to interpret functions with free variables as arguments. To that end, we require a functional logic extension of Haskell.

A very interesting application for inverse functions we identified in Haskell is that they also facilitate the implementation of advanced concepts like *functional patterns* [5], which are known from functional logic programming [6]. Functional patterns enable pattern matching at arbitrarily deep positions by allowing the use of function symbols in patterns. Using a functional pattern, the function that returns the last element of its input list can, for example, be defined as follows.

$last :: [a] \to a$
$last\ (\_ +\!\!+ [x]) = x$

In this paper, we present a framework for automatic inversion of Haskell functions. With the exception of primitive operations that interfere with the "real world", e.g., I/O operations, our automatic inversion covers the full Haskell2010 language specification [32]. In particular, we make the following contributions.

- We define a monad to model computations with free variables (Section 3). In conjunction with a standard monadic lifting (Section 2), this monad extends Haskell with the required functional logic aspects.
- We describe the automatic generation of inverse functions based on our extension of Haskell's computational model (Section 4). The inverses make use of Haskell's laziness to reduce the search space, and are, in consequence, more efficient than inverses in Prolog.
- We extend Haskell's pattern matching capabilities by adding functional patterns to the language, which we implement using inverse functions (Section 5).
- We demonstrate that our framework can also be utilized to express partial inverses, where known inputs can be fixed [47] (Section 6).
- We discuss improvements of our approach to function inversion regarding higher-order functions and numerical primitive types (Section 7).
- Last, we provide a prototype[2] of a Glasgow Haskell Compiler (GHC) plugin that performs automatic inversion as described in this paper. However, we do not cover the implementation here. The prototype also includes examples and a test suite to reinforce the correctness of our approach.

2 Monadic Lifting

In this section, we discuss the monadic lifting that stands at the core of our automatic inversion. Although Haskell uses call-by-need evaluation, our transformation is based on the call-by-name transformation as presented by Wadler [58]. This transformation, however, is only sufficient when modeling Haskell without side effects, because otherwise sharing might become observable. We take care of this issue when defining our monad for the lifting in Section 3 by incorporating sharing (of the effect) into the monad itself.

We want to keep the original definitions in the module, so that they remain usable. Therefore, we introduce our lifted definitions under a new name. To distinguish between the lifted and original definitions, the new name adds a subscript to indicate to which monad a definition has been lifted to.

2.1 Lifting of Type Expressions

Our monadic lifting of types replaces type constructors with their effectful counterparts. This includes the function type constructor $(\to)$ as well, which is replaced by the following type constructor.

$\textbf{newtype}\ (\to_m)\ a\ b = Func_m\ (m\ a \to m\ b)$

Using this type definition, the arguments of each function type constructor are wrapped in our monadic type m. Hence, both the argument and result of every lifted function are monadic. While the constraints are lifted as well, any quantifier still remains at the beginning of a type. For consistency, we also wrap the outer type of a function. The lifting of type expressions $[\![\cdot]\!]^t_m$ is presented in Figure 1. As an example, consider the following type signature and its lifted counterpart, where $[\,]_m$ is the lifted version of Haskell's list type $[\,]$.

$map :: (a \to b) \to [a] \to [b]$
$map_m :: m\ ((a \to_m b) \to_m [a]_m \to_m [b]_m)$

2.2 Lifting of Type Declarations

Because not only functions but also data constructors should comply with call-by-name, we have to lift type declarations as well. One might be tempted to apply the lifting of function types to the constructor types. For example, in the case of

$$(:) :: a \to [a] \to [a]$$

one would get the type

$$(:_m) :: m\ (a \to_m [a]_m \to_m [a]_m).$$

[1] Adapted from *The Typing Principle* by Pickering et al. [44].

[2] Available at https://github.com/cau-placc/inversion-plugin.

$$[\![\forall \alpha_1 \ldots \alpha_n.\varphi \Rightarrow \tau]\!]^{\mathsf{t}}_m \coloneqq \forall \alpha_1 \ldots \alpha_n.[\![\varphi]\!]^{\mathsf{t}}_m \Rightarrow m\ [\![\tau]\!]^{\mathsf{i}}_m \qquad \text{(Polymorphic type)}$$

$$[\![\langle T_1\ \tau_1, \ldots, T_n\ \tau_n\rangle]\!]^{\mathsf{t}}_m \coloneqq \langle [\![T_1\ \tau_1]\!]^{\mathsf{i}}_m, \ldots, [\![T_n\ \tau_n]\!]^{\mathsf{i}}_m\rangle \qquad \text{(Context)}$$

$$[\![\tau_1 \to \tau_2]\!]^{\mathsf{i}}_m \coloneqq [\![\tau_1]\!]^{\mathsf{i}}_m \to_m [\![\tau_2]\!]^{\mathsf{i}}_m \qquad \text{(Function type)}$$

$$[\![\tau_1\ \tau_2]\!]^{\mathsf{i}}_m \coloneqq [\![\tau_1]\!]^{\mathsf{i}}_m\ [\![\tau_2]\!]^{\mathsf{i}}_m \qquad \text{(Type application)}$$

$$[\![T]\!]^{\mathsf{i}}_m \coloneqq T_m \qquad \text{(Type constructor)}$$

$$[\![\alpha]\!]^{\mathsf{i}}_m \coloneqq \alpha \qquad \text{(Type variable)}$$

Figure 1. Lifting rules for type expressions parametrized over some monad m

However, the application of a constructor to a value is always defined and a constructor cannot introduce any effect to a program. Therefore, it is sufficient to just lift the arguments of each constructor, because they are the only potential sources of effects in a data type. This approach is well-known from modeling the non-strictness of Haskell, e.g., in a pure programming language like Agda [1]. The rules for the lifting of type declarations $[\![\cdot]\!]^{\mathsf{d}}_m$ are presented in Figure 2. The following code shows how the list data type [ ] is lifted.

```
data [ ]   a = [ ]
             | (:) a [a]
data [ ]_m a = [ ]_m
             | (:_m) (m a) (m [a]_m)
```

2.3 Simplifying Pattern Matching

In order to define the monadic lifting for functions, it is useful to simplify any pattern matching first because Haskell's pattern matching syntax is rich, even if we only consider Haskell2010 [32]. For the bulk of our simplification we use our own implementation of a standard pattern matching algorithm [56] with the following extensions:

1. At the beginning, we move every pattern matching into unary lambda abstractions on the right-hand side of our definition. Thus, a function definition is replaced by a constant that uses multiple unary lambda expressions to introduce the arguments of the original function. This simplifies the subsequent monadic transformation significantly. For example, the definition of (++) from Section 1 is first transformed as follows.

```
(++) = λarg1 → λarg2 → case arg1 of
  [ ]    → arg2
  x : xs → x : xs ++ arg2
```

2. Every non-exhaustive pattern matching is augmented by a catch-all pattern match failure. We use the *error* function for the failure. Later on, the lifting replaces the *error* call with a call to the monadic $fail^m$[3] operation from the class *MonadFail*. This is important for the correctness of our inverse functions when the original function is only partially defined. Thus, a suitable monad for our monadic lifting needs to be an instance of *MonadFail*.

2.4 Lifting of Functions

After function definitions have been simplified such that no argument occurs on the left-hand side, we continue by lifting the expression on the right-hand side and renaming the function accordingly. The lifted type of the function serves as a guide for lifting the expression. We can derive three rules for lifting expressions from our lifting of types:

1. As each function arrow ($\to$) is replaced by the ($\to_m$) type and wrapped in the monad m, all lambda expressions have to be wrapped in a $return^m \circ Func_m$.
2. We have to extract a value from the monad m using the corresponding bind operator ($\gg\!=^m$) before we can pattern match on it.
3. Before applying a function to an argument, we first have to extract the function from the monad m by using ($\gg\!=^m$) and pattern matching on the $Func_m$ constructor. As each function arrow is wrapped separately, this has to be done for each parameter of the original function.

In the following paragraphs we explain how the lifting works for a small subset of Haskell's syntax. Figure 3 presents the rules of our transformation $[\![\cdot]\!]^{\mathsf{e}}_m$.

Variables. A variable that denotes a top-level function, i.e., is globally scoped, has to be renamed so that it mentions the lifted definition of the function. If the variable happens to denote the *error* function, we instead replace it by $fail^m$ as mentioned earlier in Section 2.3.

Lambda Abstractions. A lambda abstraction is transformed by wrapping it in $return^m \circ Func_m$ and applying the lifting to the inner expression.

[3] We use superscripts for monadic operations to indicate which monad they operate on. For instance, $return^m$ refers to the return operation of some monad m. This notation contrasts with the subscripts we use for lifted operations.

$$\llbracket\, \textbf{data}\ T\ \alpha_1 \ldots \alpha_n = D_1 \mid \ldots \mid D_k \rrbracket^{\mathsf{d}}_m \coloneqq \textbf{data}\ T_m\ \alpha_1 \ldots \alpha_n = \llbracket D_1 \rrbracket^{\mathsf{c}}_m \mid \ldots \mid \llbracket D_k \rrbracket^{\mathsf{c}}_m \qquad \text{(Data type)}$$

$$\llbracket\, \textbf{newtype}\ T\ \alpha_1 \ldots \alpha_n = D \rrbracket^{\mathsf{d}}_m \coloneqq \textbf{newtype}\ T_m\ \alpha_1 \ldots \alpha_n = \llbracket D \rrbracket^{\mathsf{c}}_m \qquad \text{(Newtype)}$$

$$\llbracket\, \textbf{type}\ T\ \alpha_1 \ldots \alpha_n = \tau \rrbracket^{\mathsf{d}}_m \coloneqq \textbf{type}\ T_m\ \alpha_1 \ldots \alpha_n = \llbracket \tau \rrbracket^{\mathsf{i}}_m \qquad \text{(Type synonym)}$$

$$\llbracket C\ \tau_1 \ldots \tau_n \rrbracket^{\mathsf{c}}_m \coloneqq C_m\ \llbracket \tau_1 \rrbracket^{\mathsf{t}}_m \ldots \llbracket \tau_n \rrbracket^{\mathsf{t}}_m \qquad \text{(Constructor)}$$

Figure 2. Lifting rules for type declarations parametrized over some monad m

$$\llbracket v \rrbracket^{\mathsf{e}}_m \coloneqq \begin{cases} return^m \circ Func_m \circ (\ggg^m fail^m) & \text{if } v \text{ denotes the function } error \\ v_m & \text{if } v \text{ is globally scoped} \\ v & \text{otherwise} \end{cases} \qquad \text{(Variable)}$$

$$\llbracket \lambda x \to e \rrbracket^{\mathsf{e}}_m \coloneqq (return^m \circ Func_m)\ (\lambda x \to \llbracket e \rrbracket^{\mathsf{e}}_m) \qquad \text{(Abstraction)}$$

$$\llbracket e_1\ e_2 \rrbracket^{\mathsf{e}}_m \coloneqq \llbracket e_1 \rrbracket^{\mathsf{e}}_m\ \text{`}app^m\text{`}\ \llbracket e_2 \rrbracket^{\mathsf{e}}_m \qquad \text{(Application)}$$

$$\begin{aligned}\llbracket C \rrbracket^{\mathsf{e}}_m \coloneqq{} & (return^m \circ Func_m)\ (\lambda y_1 \to \ldots \to (return^m \circ Func_m)\ (\lambda y_n \to \\ & return^m\ (C_m\ y_1 \ldots y_n))) \qquad \text{where } C \text{ has arity } n \end{aligned} \qquad \text{(Constructor)}$$

$$\llbracket\, \textbf{case}\ e\ \textbf{of}\ \{ br_1; \ldots; br_n \} \rrbracket^{\mathsf{e}}_m \coloneqq \llbracket e \rrbracket^{\mathsf{e}}_m \ggg^m \lambda\textbf{case}\ \{ \llbracket br_1 \rrbracket^{\mathsf{b}}_m; \ldots; \llbracket br_n \rrbracket^{\mathsf{b}}_m \} \qquad \text{(Case expression)}$$

$$\llbracket C\ x_1 \ldots x_n \to e \rrbracket^{\mathsf{b}}_m \coloneqq C_m\ x_1 \ldots x_n \to \llbracket e \rrbracket^{\mathsf{e}}_m \qquad \text{(Case branch)}$$

Figure 3. Lifting rules for expressions parametrized over some monad m (y_1 to y_n are fresh variables)

Applications. We extract the "real" function from the monad before applying it in the lifted setting using the following helper function, which we will use as a left-associative infix operator.

```
infixl `app^m`
app^m :: Monad m ⇒ m (a →_m b) → m a → m b
mf `app^m` mx = mf ≫=^m λ(Func_m f) → f mx
```

An application of a function to more than one argument is represented as multiple nested applications.

Constructors. While a function of type

$$\tau_1 \to \ldots \to \tau_n \to \tau'$$

is transformed into a function of type

$$m\ (\llbracket \tau_1 \rrbracket^{\mathsf{i}}_m \to_m \ldots \to_m \llbracket \tau_n \rrbracket^{\mathsf{i}}_m \to_m \llbracket \tau' \rrbracket^{\mathsf{i}}_m),$$

a similar constructor type is transformed into

$$\llbracket \tau_1 \rrbracket^{\mathsf{t}}_m \to \ldots \to \llbracket \tau_n \rrbracket^{\mathsf{t}}_m \to \llbracket \tau' \rrbracket^{\mathsf{i}}_m.$$

As applications of expressions in Haskell do not differentiate between functions and constructors, we need to solve this type discrepancy by transforming a value constructor occurring in expressions into a nested chain of $return^m \circ Func_m$ applications and lambda abstractions.

Case Expressions. Before performing a case analysis on a term in the lifted setting, we need to extract the value from the effect monad by means of $(\ggg^m)$.

To conclude the lifting of functions with an example, we revisit the function (++), whose simplified variant was already shown in Section 2.3. The following code illustrates what the lifted version of that function looks like in the end.

```
(++_m) :: Monad m ⇒ m ([a]_m →_m [a]_m →_m [a]_m)
(++_m) = (return^m ∘ Func_m) $ λarg1 →
         (return^m ∘ Func_m) $ λarg2 → arg1 ≫=^m λcase
  []_m    → arg2
  x :_m xs → ((return^m ∘ Func_m) $ λy_1 →
               (return^m ∘ Func_m) $ λy_2 →
                 return^m (y_1 :_m y_2))
    `app^m` x `app^m` ((++_m) `app^m` xs `app^m` arg2)
```

2.5 Lifting of Type Classes

Type (constructor) classes [26, 59] are renamed for their lifting. Their methods and default implementations are lifted just like regular functions. Instances are lifted similarly. For example, the following code shows how a simplified *Eq* class is lifted.

```
class Eq a where
  (==) :: a → a → Bool
class Eq_m a where
  (==_m) :: m (a →_m a →_m Bool_m)
```

3 Effect Monad

With the monadic lifting in mind, we can proceed with the definition of the actual monad that we use for the lifting. Recall from Section 1 that we base our automatic inversion on a functional logic extension of Haskell by non-determinism. We will later see that the non-determinism originates solely from the instantiation of free variables. Consequently, our effect monad must be able to model non-deterministic computations as well as free variables.

We do not define a single monad but two nested monads instead. Using nested monads enables both implicit and explicit handling of free variables as needed. The inner monad deals with the non-determinism effect, whereas the outer monad models computations with the implicit instantiation of free variables.

3.1 Non-Determinism Monad

While a monad that is an instance of *MonadPlus*, e.g., the list monad, usually suffices to model non-determinism [55], in our case, we additionally require a state to account for the potential sharing of the non-determinism effect [15]. The issue of sharing non-determinism stems from Haskell's call-by-need evaluation. In the context of functional logic programming, the combination of non-determinism and call-by-need evaluation is known as call-time choice [21]. Because any occurring non-determinism will originate solely from the instantiation of free variables, it is sufficient to memorize the value that a free variable has been instantiated with for each computation branch. This fact also justifies the use of the call-by-name transformation (see Section 2), because we explicitly share the part that introduces effects, namely the free variables, by means of our state.

To memorize the bindings of free variables, we use an untyped[4] heap that maps free variables to their instantiated values. In order to identify free variables, we use the following type synonym.

type *ID* = *Integer*

The heap is given by the following abstract data type.

type *Heap*
empty :: *Heap*
insert :: *ID* → *a* → *Heap* → *Heap*
find :: *ID* → *Heap* → *Maybe a*

The function *empty* creates an empty heap, *insert* adds a binding for an identifier to an existing heap, and *find* returns the value for an identifier from the heap if a binding for that identifier exists on the heap.

For the definition of our stateful non-determinism monad we use the state monad transformer[5] [27, 30]. We apply the transformer to a monad that is an instance of *MonadPlus*. More precisely, we use an efficient implementation of a tree-based monad, namely *Search*[6], that allows for the application of different search strategies. In addition to the heap, the state contains an identifier that represents the next available identifier. This identifier will become relevant for newly generated free variables when instantiating free variables (see Section 3.2). We end up with the following definition.

type *ND a* = *StateT* (*Heap*, *ID*) *Search a*

Finally, we provide a monad starter. It runs a stateful non-deterministic computation with an initially empty heap and 0 as the first available identifier. The monad starter returns the results of the non-deterministic computation in a list, which is obtained using the function *bfs* that traverses the *Search* structure breadth-first. We use breadth-first search because of its completeness property.

evalND :: *ND a* → [*a*]
evalND nd = *bfs* (*evalStateT nd* (*empty*, 0))

3.2 Functional Logic Monad

In order to define the monad for functional logic computations with free variables, we first need a concrete representation of free variables on the value level. To this end, we introduce the following data type that distinguishes between free variables and other values.

data *FLVal a* = *Var ID*
  | *Val* {*unVal* :: *a*}

Functional logic computations with free variables are non-deterministic computations that operate on the data type *FLVal*, which is expressed by the following type that builds on top of the non-determinism monad.

newtype *FL a* = *FL* {*unFL* :: *ND* (*FLVal a*)}

Nesting the non-determinism monad within the functional logic monad allows us to easily work with the explicit representation of free variables if needed. We exploit this aspect, for example, in Section 4.2.

Next, we give the *Monad* instance[7] for the *FL* type and start by defining the $return^{FL}$ operation.

$return^{FL}$ :: *a* → *FL a*
$return^{FL}$ *x* = *FL* ($return^{ND}$ (*Val x*))

Alongside, we introduce a function *freeFL* that returns a free variable in the functional logic monad.

freeFL :: *ID* → *FL a*
freeFL i = *FL* ($return^{ND}$ (*Var i*))

[4] For the sake of simplicity, we forego a type-safe solution based on the *Typeable* class [43]. Doing so is unproblematic, because the heap is never exposed to the user.
[5] Available at https://hackage.haskell.org/package/transformers.
[6] Available at https://hackage.haskell.org/package/tree-monad.
[7] To be exact, *FL* is a constrained monad [23, 49] due to a constraint on the bind operation. Furthermore, *FL* is only a monad w.r.t. *run equality* [25], i.e., the monad laws hold after applying our later defined monad starter to both sides of the equations.

The bind operation, however, is more challenging to define. Our idea is that pattern matching should lead to the instantiation of free variables. This approach corresponds to the concept of narrowing [45] known from functional logic programming. Since the monadic lifting from Section 2.4 transforms pattern matching into monadic binds, the bind operation of our functional logic monad has to perform the instantiation.

To instantiate free variables, we have to be able to enumerate all constructors of the corresponding data types. For this purpose, we introduce the type class *Narrowable*. Its only method *narrow* is supposed to get the next available identifier from the state of the non-determinism monad (see Section 3.1) as an argument. Using this identifier, it should provide a list of all constructors of a data type applied to freshly generated free variables. Additionally, *narrow* should return the number of identifiers used for each constructor, which corresponds exactly to the constructor's arity.

$$\begin{array}{l}\textbf{class}\ \mathit{Narrowable}\ a\ \textbf{where}\\ \quad \mathit{narrow} :: \mathit{ID} \to [(a, \mathit{Integer})]\end{array}$$

Instances of the *Narrowable* class are defined for *FL*-lifted data types. As an example, consider the following instance for the lifted list data type $[\,]_{FL}$.

$$\begin{array}{l}\textbf{instance}\ \mathit{Narrowable}\ [a]_{FL}\ \textbf{where}\\ \quad \mathit{narrow}\ i = [([\,]_{FL}, 0), (\mathit{freeFL}\ i :_{FL} \mathit{freeFL}\ (i+1), 2)]\end{array}$$

Before we finally define the bind operation of our functional logic monad, we introduce the following auxiliary function in the non-determinism monad.

$$\begin{array}{l}\mathit{instantiateND} :: \mathit{Narrowable}\ a \Rightarrow \mathit{ID} \to \mathit{ND}\ a\\ \mathit{instantiateND}\ i = \mathit{get} \gg\!=^{ND} \lambda(h, j) \to \textbf{case}\ \mathit{find}\ i\ h\ \textbf{of}\\ \quad \mathit{Nothing} \to \mathit{msum}^{ND}\ (\mathit{map}\ \mathit{update}\ (\mathit{narrow}\ j))\\ \qquad \textbf{where}\ \mathit{update}\ (x, o) = \mathit{put}\ (\mathit{insert}\ i\ x\ h, j + o) \gg^{ND}\\ \qquad\qquad\qquad\qquad\qquad\qquad \mathit{return}^{ND}\ x\\ \quad \mathit{Just}\ x \quad\ \to \mathit{return}^{ND}\ x\end{array}$$

This function tries to find a binding for the variable on the heap. If such a binding exists, which would mean that the variable has been instantiated before, we return the value it is bound to. Note that this portion of code implements call-time choice semantics, i.e., free variables with the same identifier represent the same value within the same computation branch. If there is no binding on the heap for the variable, we generate the list of possible constructors for said variable using *narrow*. Every element of this list leads to a new computation branch, in which we put the value on the heap, increment the next available identifier of the non-determinism monad's state accordingly, and return the value.

Now we have everything at hand to define the bind operation for the *FL* monad. By binding the inner non-deterministic computation, we can pattern match on its result. In the case that the value is a free variable, we instantiate it using the previously introduced auxiliary function *instantiateND* and apply the continuation to the instantiated value. Otherwise, we directly apply the continuation to the value.

$$\begin{array}{l}(\gg\!=^{FL}) :: \mathit{Narrowable}\ a \Rightarrow \mathit{FL}\ a \to (a \to \mathit{FL}\ b) \to \mathit{FL}\ b\\ \mathit{FL}\ \mathit{nd} \gg\!=^{FL} f = \mathit{FL}\ \$\ \mathit{nd} \gg\!=^{ND} \lambda\textbf{case}\\ \quad \mathit{Var}\ i \to \mathit{instantiateND}\ i \gg\!=^{ND} \mathit{unFL} \circ f\\ \quad \mathit{Val}\ x \to \mathit{unFL}\ (f\ x)\end{array}$$

Due to the transformation of function applications into calls of app^{FL} (or rather $(\gg\!=^{FL})$) during the lifting of functions (see Section 2.4), we have to give a *Narrowable* instance for the lifted function type $(\to_{FL})$. Since we are unable to generate arbitrary functions, we resort to a run-time error in this case. However, this error is irrelevant in practice because narrowing of free variables with function types can never be triggered as we restrict the usage of inverses involving higher-order types (see Section 7.1).

$$\begin{array}{l}\textbf{instance}\ \mathit{Narrowable}\ (a \to_{FL} b)\ \textbf{where}\\ \quad \mathit{narrow}\ \_ = \mathit{error}\ \texttt{"cannot narrow functions"}\end{array}$$

Remember that *FL* has to be an instance of *MonadFail* in order to be a suitable monad for the monadic lifting as presented in Section 2.3. For the instance definition we fall back to the instance of the inner non-determinism monad.

$$\begin{array}{l}\mathit{fail}^{FL} :: \mathit{String} \to \mathit{FL}\ a\\ \mathit{fail}^{FL}\ s = \mathit{FL}\ (\mathit{fail}^{ND}\ s)\end{array}$$

We further introduce the following shorthand function to represent failed computations without error messages.

$$\begin{array}{l}\mathit{failedFL} :: \mathit{FL}\ a\\ \mathit{failedFL} = \mathit{FL}\ \mathit{mzero}^{ND}\end{array}$$

Lastly, we need a monad starter to run a functional logic computation. The result of a functional logic computation might contain free variables (or even be a free variable itself), which is generally undesired after executing a functional logic computation. After running the monad starter, we expect values to be effect-free, i.e., the results of a functional logic computation must not contain any free variables.

To this end, we introduce the *Groundable* type class. Like *Narrowable*, it operates on *FL*-lifted values. Its single method *groundFL* should instantiate any deep occurring free variables to every possible value and, thus, should compute the ground normal form of its argument. This invariant will become relevant, for instance, in Section 4.1.

$$\begin{array}{l}\textbf{class}\ \mathit{Narrowable}\ a \Rightarrow \mathit{Groundable}\ a\ \textbf{where}\\ \quad \mathit{groundFL} :: \mathit{FL}\ a \to \mathit{FL}\ a\end{array}$$

As an example, we once again have a look at the instance definition for the lifted list data type $[\,]_{FL}$. In the instance, we simply call *groundFL* recursively on every component.

$$
\begin{array}{l}
\textbf{instance } \mathit{Groundable}\ a \Rightarrow \mathit{Groundable}\ [\,a\,]_{FL}\ \textbf{where} \\
\quad \mathit{groundFL}\ x = x \ggg^{FL} \lambda\textbf{case} \\
\quad\quad [\,]_{FL} \quad\ \ \rightarrow \mathit{return}^{FL}\ [\,]_{FL} \\
\quad\quad y :_{FL} ys \rightarrow \mathit{groundFL}\ y \ggg^{FL} \lambda y' \rightarrow \\
\quad\quad\quad\quad\quad\quad \mathit{groundFL}\ ys \ggg^{FL} \lambda ys' \rightarrow \\
\quad\quad\quad\quad\quad\quad \mathit{return}^{FL}\ (\mathit{return}^{FL}\ y' :_{FL} \mathit{return}^{FL}\ ys')
\end{array}
$$

The final monad starter for a functional logic computation looks as follows. It first computes the ground normal form of its argument, then runs the monad starter for the inner non-deterministic computation and last strips the *Val* constructor of all results using *unVal*. The latter is safe to do because of the ground normal form computation beforehand.

$$
\begin{array}{l}
\mathit{evalFL} :: \mathit{Groundable}\ a \Rightarrow \mathit{FL}\ a \rightarrow [\,a\,] \\
\mathit{evalFL} = \mathit{map}\ \mathit{unVal} \circ \mathit{evalND} \circ \mathit{unFL} \circ \mathit{groundFL}
\end{array}
$$

4 Inverses

In this section, we describe the generation of inverse functions. Our general idea for an inverse function is to interpret the monadically lifted version of a function using free variables as the arguments. For the lifting, we use the functional logic monad from Section 3.2. The evaluation of the lifted function then leads to the non-deterministic instantiation of the free variables if the original function performed pattern matching on the corresponding values. Recall from Section 2.4 that pattern matching has been transformed into calls to $(\ggg^{FL})$, which implicitly instantiates free variables and creates multiple non-deterministic computation branches in the process. Finally, each non-deterministic result of the lifted function call is matched with the argument to the inverse function. Within each matching computation branch, the heap will contain bindings for the free variables that represent the result of the inverse computation.

4.1 Conversion

So far we have introduced a lifted variant for every data type and function in our program. Our lifted functions operate on the lifted data types. However, the generated inverse functions should take the original unlifted types as arguments and return unlifted representations as well. Thus, we need to convert between regular and lifted representations of data types for interoperability between the lifted world and the regular Haskell world.

First, we introduce the following poly-kinded [60] type family [48] in order to use the lifting of types within Haskell itself. In fact, the type family *Lifted* exactly corresponds to $[\![\cdot]\!]^{\mathrm{i}}_{FL}$ as depicted in Figure 1.

$$\textbf{type family } \mathit{Lifted}\ (a :: k) :: k$$

Furthermore, for every lifted data type or type class T, we generate a type family instance of the form

$$\textbf{type instance } \mathit{Lifted}\ T = T_{FL},$$

for which the following instance is an example.

$$\textbf{type instance } \mathit{Lifted}\ [\,] = [\,]_{FL}$$

To fully model the type lifting, we additionally need a type family instance that represents the lifting of an application of two type expressions.[8]

$$\textbf{type instance } \mathit{Lifted}\ (f\ \ a) = (\mathit{Lifted}\ f)\ \ (\mathit{Lifted}\ a)$$

Next, we define the class *Convertible*, which performs the actual conversion between regular and lifted values.

$$
\begin{array}{l}
\textbf{class } \mathit{Convertible}\ a\ \textbf{where} \\
\quad \mathit{to} \quad\ :: a \rightarrow \mathit{Lifted}\ a \\
\quad \mathit{from} :: \mathit{Lifted}\ a \rightarrow a
\end{array}
$$

Instances of the *Convertible* type class are quite simple to define, because they just map every data constructor to their lifted variant and vice versa. This can be seen in the following instance for the list data type [ ].

$$
\begin{array}{l}
\textbf{instance } \mathit{Convertible}\ a \Rightarrow \mathit{Convertible}\ [\,a\,]\ \textbf{where} \\
\quad \mathit{to}\ [\,] \qquad\quad = [\,]_{FL} \\
\quad \mathit{to}\ (x : xs) = \mathit{toFL}\ x :_{FL} \mathit{toFL}\ xs \\
\quad \mathit{from}\ [\,]_{FL} \qquad = [\,] \\
\quad \mathit{from}\ (x :_{FL} xs) = \mathit{fromFL}\ x : \mathit{fromFL}\ xs
\end{array}
$$

For convenience, we use two helper functions in the instance above that handle the conversion of components of data constructors, namely *toFL* and *fromFL*. The function *toFL* converts its unlifted argument using *to* and then lifts the result into the *FL* monad with return^{FL}.

$$
\begin{array}{l}
\mathit{toFL} :: \mathit{Convertible}\ a \Rightarrow a \rightarrow \mathit{FL}\ (\mathit{Lifted}\ a) \\
\mathit{toFL} = \mathit{return}^{FL} \circ \mathit{to}
\end{array}
$$

The definition of *fromFL*, which translates from the lifted world back to the Haskell world, is a bit more elaborate. We will see in Section 4.3 that conversion in this direction only takes place after the monad starter ran. Thus, the argument to *from* and *fromFL* will always be in ground normal form. Consequently, it is safe to just extract the single value from the inner non-deterministic computation and to recursively convert it.

$$
\begin{array}{l}
\mathit{fromFL} :: \mathit{Convertible}\ a \Rightarrow \mathit{FL}\ (\mathit{Lifted}\ a) \rightarrow a \\
\mathit{fromFL} = \mathit{from} \circ \mathit{unVal} \circ \mathit{head} \circ \mathit{evalND} \circ \mathit{unFL}
\end{array}
$$

4.2 Matching

We must ensure that the argument of an inverse function matches the result of the computation of the lifted function. Our matching procedure consists of two intertwined parts.

The first part is captured by the type class *Matchable* with a monadic function *match* that ensures that the topmost constructors of its two arguments correspond. The method

[8] Note that – although our type lifting is injective – we cannot declare the type family *Lifted* to be injective [54], because the instance for type applications would otherwise be rejected by the GHC.

should add bindings to the heap on success or fail otherwise. Its first argument is unlifted, because it will always be supplied from the Haskell world as we will see in Section 4.3.

$$\begin{array}{l}\textbf{class}\ \mathit{Matchable}\ a\ \textbf{where}\\ \quad \mathit{match} :: a \rightarrow \mathit{Lifted}\ a \rightarrow \mathit{FL}\ ()_{FL}\end{array}$$

Instances of this class are easily implemented for any data type. For every pair of unlifted and lifted constructor we pair-wise match all their components. If the constructors do not match, we fail using *failedFL* from Section 3.2.

$$\begin{array}{l}\textbf{instance}\ (\mathit{Convertible}\ a, \mathit{Matchable}\ a) \Rightarrow\\ \qquad\qquad \mathit{Matchable}\ [a]\ \textbf{where}\\ \quad \mathit{match}\ [\,]\qquad [\,]_{FL} \qquad = \mathit{return}^{FL}\ ()_{FL}\\ \quad \mathit{match}\ (x:xs)\ (y:_{FL} ys) = \mathit{matchFL}\ x\ y \gg^{FL}\\ \qquad\qquad\qquad\qquad\qquad \mathit{matchFL}\ xs\ ys\\ \quad \mathit{match}\ \_ \qquad \_ \qquad\quad = \mathit{failedFL}\end{array}$$

Since the components of a lifted data type are values in the functional logic monad, the code above uses another monadic function named *matchFL*, which constitutes the second part of our matching. While it also takes an unlifted value from the Haskell world as the first argument, its second argument is a functional logic computation. In order to efficiently handle the special case that the result of this computation is a variable, we implement *matchFL* on the level of the non-determinism monad (which motivated the two-layer design of our monad in Section 3.2). We extract the result of the inner non-deterministic computation using ($\gg=^{ND}$) and afterwards consider two cases.

1. If the result is an unbound variable, the matching adds a binding for the variable on the heap and succeeds.
2. For every other case, we continue the matching with the result value using the previously defined *match*.

$$\begin{array}{l}\mathit{matchFL} :: (\mathit{Convertible}\ a, \mathit{Matchable}\ a)\\ \qquad\quad \Rightarrow a \rightarrow \mathit{FL}\ (\mathit{Lifted}\ a) \rightarrow \mathit{FL}\ ()_{FL}\\ \mathit{matchFL}\ x\ (\mathit{FL}\ nd) = \mathit{FL}\ \$\ nd \gg=^{ND} \lambda\textbf{case}\\ \quad \mathit{Var}\ i \rightarrow \mathit{get} \gg=^{ND} \lambda(h,j) \rightarrow \textbf{case}\ \mathit{find}\ i\ h\ \textbf{of}\\ \qquad \mathit{Nothing} \rightarrow \mathit{put}\ (\mathit{insert}\ i\ (\mathit{to}\ x)\ h, j) \gg^{ND}\\ \qquad\qquad\quad \mathit{return}^{ND}\ (\mathit{Val}\ ()_{FL})\\ \qquad \mathit{Just}\ y \quad \rightarrow \mathit{unFL}\ (\mathit{match}\ x\ y)\\ \quad \mathit{Val}\ y \rightarrow \mathit{unFL}\ (\mathit{match}\ x\ y)\end{array}$$

4.3 Synthesizing Inverse Functions

The last step is to synthesize the definition of a function's inverse. With all the building blocks at hand, we only need to assemble them in the right manner. Since the type signature will become more clear after seeing the implementation, we start with the latter.

In the following, we explain the implementation of inverse functions by the example of $(+\!\!+)^{-1}$ from Section 1. In the implementation, we call the lifted function ($+\!\!+_{FL}$) with free variables as arguments. We use negative identifiers for the free variables to rule out clashes with the positive identifiers that are generated during the instantiation of free variables (see Section 3.2). The result of the lifted function call is matched with the input of the inverse function using *matchFL*. Every computation branch for which the matching succeeds results in a heap that contains bindings for free variables that were demanded during the computation. Note that unsuccessful matchings fail as fast as possible due to Haskell's laziness, because evaluation of the lifted function is only required up to the first mismatching constructors. In each remaining computation branch, we return a (lifted) tuple of the free variables we initially used as the arguments to the lifted function. The free variables may have been (partially) instantiated during the computation of the lifted function and will be fully instantiated by the monad starter *evalFL* when it computes the ground normal form. All that is left is to convert the values in ground normal form back into their unlifted representation via *map* and *from*.

$$\begin{array}{l}(+\!\!+)^{-1}\ \mathit{res} = \mathit{map}\ \mathit{from}\ \$\ \mathit{evalFL}\ \$\\ \quad \mathit{matchFL}\ \mathit{res}\ ((+\!\!+_{FL})\ \text{`}\mathit{app}^{FL}\text{'}\ \mathit{freeFL}\ (-1)\\ \qquad\qquad\qquad\qquad \text{`}\mathit{app}^{FL}\text{'}\ \mathit{freeFL}\ (-2)) \gg^{FL}\\ \quad \mathit{return}^{FL}\ ((,)_{FL}\ (\mathit{freeFL}\ (-1))\ (\mathit{freeFL}\ (-2)))\end{array}$$

Now that we have seen the implementation of the inverse, we can turn our attention to its type signature. The type signature of $(+\!\!+)^{-1}$ has to accommodate that we need to match the (unlifted) argument of the inverse function with the result of our lifted function. Furthermore, the result lists of the inverse function need to be converted from their lifted representation back into the Haskell world. And lastly, due to the use of the monad starter, we demand *Groundable* for the lifted lists as well. We end up with the following type signature, which requires the use of the *FlexibleContexts* language extension.

$$\begin{array}{l}(+\!\!+)^{-1} :: (\mathit{Matchable}\ [a], \mathit{Convertible}\ [a]\\ \qquad\qquad , \mathit{Groundable}\ (\mathit{Lifted}\ [a]))\\ \qquad\quad \Rightarrow [a] \rightarrow [([a],[a])]\end{array}$$

Since every data type[9] has instances of both *Matchable* and *Convertible* as well as a *Groundable* instance on its lifted variant, we introduce the following constraint synonym using the *ConstraintKinds* language extension [40]. We use it to increase readability of type signatures—except when a more granular context is required, e.g., when we consider partial inverses in Section 6.

$$\begin{array}{l}\textbf{type}\ \mathit{Invertible}\ a =\\ \quad (\mathit{Matchable}\ a, \mathit{Convertible}\ a, \mathit{Groundable}\ (\mathit{Lifted}\ a))\end{array}$$

Using this constraint synonym along with simplifying the constraints using the available instances, we can write the type signature as follows.

$$(+\!\!+)^{-1} :: \mathit{Invertible}\ a \Rightarrow [a] \rightarrow [([a],[a])]$$

[9] With the exception of the function type, see Section 7.1.

5 Functional Patterns

Functional patterns [5, 6] enable arbitrarily deep pattern matching in data structures by allowing the use of function symbols in patterns. In this section, we implement functional patterns in Haskell using inverse functions and enrich Haskell's pattern matching this way.

Recall the following example from Section 1, where we use the function (++) in a pattern to specify that the variable x should correspond to the last element of the input list.

$$\mathit{last}\ (\_ + [x]) = x$$

In Curry [20], functional patterns conceptually represent a (potentially infinite) set of rules. In the case of the function *last*, the functional pattern is equivalent to the following infinite number of equations.

$$\begin{aligned} &\mathit{last}\ [x] &&= x \\ &\mathit{last}\ [\_, x] &&= x \\ &\mathit{last}\ [\_, \_, x] &&= x \\ &\dots \end{aligned}$$

In general, a functional pattern can match in multiple ways. That is, functional patterns are per se non-deterministic.[10] In order to properly integrate functional patterns with Haskell where functions are deterministic, we expect them to have a single or multiple equivalent solutions. We leave it to the programmer to ensure this property. Without this property, the semantics of functional patterns in Haskell are not well-defined, i.e., the solution of a functional pattern could be any of all non-deterministic solutions.

Assuming that the aforementioned property holds, we can define a pattern transformation $(\!|\cdot|\!)^{\mathsf{funPat}}$ that expresses functional patterns in terms of inverse functions and view patterns [14, 57]. The transformation is given in Figure 4 and is available as a Template Haskell [31, 52] meta function in our GHC plugin. Due to the nature of the transformation to create regular Haskell patterns, functional patterns integrate well with Haskell's already existing pattern matching: They are allowed to occur in place of any pattern and can even be nested within another. Furthermore, in accordance with Haskell's pattern matching semantics, a function's remaining rules are tried if a functional pattern does not match. Using the transformation we can then write the function *last* as follows.

$$\mathit{last}\ (\!|\_ + [x]|\!)^{\mathsf{funPat}} = x$$

We now explain how the functional pattern transformation works by means of the example *last*, whose final transformed version looks as follows.

$$\begin{aligned} &\mathit{last} :: \mathit{Invertible}\ a \Rightarrow [a] \to a \\ &\mathit{last}\ ((\lambda \mathit{arg} \to [\mathit{res} \mid \mathit{res}@(\_, [x]) \leftarrow (+\!\!+)^{-1}\ \mathit{arg}]) \\ &\qquad \to (\_, [x]) : \_) = x \end{aligned}$$

[10] Although this is not the case for *last*, because a partially applied (++) is injective.

In the function of the view pattern, we first call the inverse $(+\!\!+)^{-1}$. The results of this call are then filtered with the list comprehension

$$[\mathit{res} \mid \mathit{res}@(\_, [x]) \leftarrow (+\!\!+)^{-1}\ \mathit{arg}]$$

so that they match the argument patterns given in the functional pattern. In the view pattern's pattern

$$(\_, [x]) : \_$$

we reuse the argument patterns to bind the functional pattern's variables accordingly. Here, it is safe to match only on the first result in the view, because we assume the single-solution property to hold. Note the additionally imposed *Invertible* constraint on the type variable a in the type signature of *last* due to the use of $(+\!\!+)^{-1}$.

The run time of *last* as presented is quadratic in the length of the input list. However, we achieve linear run time if we omit the evaluation to ground normal form (see Section 3.2) for functional patterns. We claim that it is safe to do so for inverses in functional patterns, because ground normal form computation only makes a difference for inverses where the result still contains free variables. And since free variables generally represent multiple values, such inverses would violate the single-solution property. Therefore, our GHC plugin incorporates this improvement.

As a side note, we can also use non-right-linear functions in a functional pattern and, thus, render non-linear pattern matching in Haskell possible. In the following example, the functional pattern in the first equation of *isSame* is equivalent to the non-linear pattern (x, x).

$$\begin{aligned} &\mathit{dup} :: a \to (a, a) \\ &\mathit{dup}\ x = (x, x) \\ &\mathit{isSame} :: \mathit{Invertible}\ a \Rightarrow (a, a) \to \mathit{Bool} \\ &\mathit{isSame}\ (\!|\mathit{dup}\ x|\!)^{\mathsf{funPat}} = \mathit{True} \\ &\mathit{isSame}\ \_ \qquad\qquad\quad = \mathit{False} \end{aligned}$$

6 Partial Inverses

We can generalize our approach of function inversion to support partial inverses [47] as well. Partial inverses allow fixing known arguments of the original function, which is often useful. It is possible to fix a single argument, multiple arguments or none at all. Fixing no arguments corresponds to the inverses as presented in Section 4. We denote the set of indices of fixed arguments (starting at 1) as a subscript to the inverse function.

As an example, we consider the function *lookup*. Taking a key as well as a list of key-value pairs, it yields the first corresponding value if the key is present in the list.

$$\begin{aligned} &\mathit{lookup} :: \mathit{Eq}\ a \Rightarrow a \to [(a, b)] \to \mathit{Maybe}\ b \\ &\mathit{lookup}\ \_\ [\,] &&= \mathit{Nothing} \\ &\mathit{lookup}\ k\ ((k', v) : \mathit{kvs}) \mid k == k' &&= \mathit{Just}\ v \\ &\qquad\qquad\qquad\qquad\quad \mid \mathit{otherwise} &&= \mathit{lookup}\ k\ \mathit{kvs} \end{aligned}$$

$$(\!|f\ p_1 \ldots p_n|\!)^{\mathrm{funPat}} := (\lambda y_1 \rightarrow [\, y_2 \mid y_2@(p_1, \ldots, p_n) \leftarrow f^{-1}\ y_1 \,]) \rightarrow (p_1, \ldots, p_n) : \_ \qquad \text{(Functional pattern)}$$

Figure 4. Transformation of functional patterns (y_1 and y_2 are fresh variables)

Conceptually, its inverse function $lookup^{-1}$ enumerates all tuples of keys and lists of key-value pairs with the following conditions: If the input is a *Just* value, for each result tuple the list of key-value pairs has to contain at least one entry for the key. Otherwise, if the input is *Nothing*, the result keys must not be present in the corresponding result lists. However, such an inverse function is of little use in practice.

$$\begin{aligned} lookup^{-1} &:: (Eq\ a, Lifted\ (Eq\ a), Invertible\ a, Invertible\ b) \\ &\Rightarrow Maybe\ b \rightarrow [\,(a, [\,(a, b)\,])\,] \end{aligned}$$

In contrast, a partial inverse that fixes the second argument[11], namely the key-value list, is much more convenient and resembles a "reverse lookup". Instead of returning a list of key-value pairs it now expects such a list as another input in addition to the *Maybe* value. We generate the partial inverse similarly to the regular inverse, but provide the fixed argument to the lifted function (converted using *toFL* from Section 4.1). Note that we require neither ground normal form computation nor matching for the fixed argument, which is why a *Convertible* constraint is sufficient for it.

$$\begin{aligned} lookup^{-1}_{\{2\}} &:: (Eq\ a, Lifted\ (Eq\ a), Convertible\ [\,(a, b)\,] \\ &\quad , Invertible\ (Maybe\ b), Invertible\ [\,a\,]) \\ &\Rightarrow [\,(a, b)\,] \rightarrow Maybe\ b \rightarrow [\,a\,] \end{aligned}$$

$$\begin{aligned} & lookup^{-1}_{\{2\}}\ arg2\ res = map\ from\ \$\ evalFL\ \$ \\ &\quad matchFL\ res\ (lookupFL\ \text{`}app^{FL}\text{`}\ freeFL\ (-1) \\ &\qquad\qquad\qquad\qquad \text{`}app^{FL}\text{`}\ toFL\ arg2) \ggg^{FL} \\ &\quad freeFL\ (-1) \end{aligned}$$

Analog to Section 4.3, the type signature can be simplified using the available type class instances.

$$\begin{aligned} lookup^{-1}_{\{2\}} &:: (Eq\ a, Lifted\ (Eq\ a), Invertible\ a, Invertible\ b) \\ &\Rightarrow [\,(a, b)\,] \rightarrow Maybe\ b \rightarrow [\,a\,] \end{aligned}$$

To conclude this section, we demonstrate the usefulness of the partial inverse $lookup^{-1}_{\{2\}}$ with the following use cases. The first one is to get all keys in a given key-value list that map to a certain value.

ghci> $lookup^{-1}_{\{2\}}\ [\,(0, True), (2, True)\,]\ (Just\ True)$
$[\,2, 0\,]$
ghci> $lookup^{-1}_{\{2\}}\ [\,(0, True), (2, True)\,]\ (Just\ False)$
$[\,]$

The second use case of interest is to retrieve all keys that are not present in a key-value list.

ghci> $take\ 5\ (lookup^{-1}_{\{2\}}\ [\,(0, True), (2, True)\,]\ Nothing)$
$[\,1, -1, -2, 3, -3\,]$

[11] We denote the set of indices of fixed argument as a subscript.

7 Extensions

Many other approaches to function inversion are limited to work only in a first-order setting or do not cover the support of primitive types. In this section, we extend our framework with respect to both these aspects.

7.1 Higher-Order Functions

Up to this point, higher-order functions are only supported to a limited extent. (Partially) inverted functions may use higher-order functions, but must not be higher-order functions themselves. That is, inverted functions must not have higher-order arguments or return values. This restriction is ensured at type-level by the lack of instances able to satisfy *Invertible* constraints for the function type.

As an example, we consider the well-known *map* function, whose inverse has the following type (without simplifying the constraints using the available instances).

$$\begin{aligned} map^{-1} &:: (Invertible\ (a \rightarrow b), Invertible\ [\,a\,], Invertible\ [\,b\,]) \\ &\Rightarrow [\,b\,] \rightarrow [\,(a \rightarrow b, [\,a\,])\,] \end{aligned}$$

While the definition of map^{-1} is synthesized and type checks, the function cannot actually be used due to the unsatisfiable *Invertible* $(a \rightarrow b)$ constraint. Providing the missing instances could be considered problematic, because we would need to be able to narrow and match functions. This restriction is also present in some functional logic programming languages, where free variables and unification for function types are prohibited [19].

However, if we consider the partial inverse $map^{-1}_{\{1\}}$, where we fix the function argument, we only need a *Convertible* instance for the function type according to Section 6.

$$\begin{aligned} map^{-1}_{\{1\}} &:: (Convertible\ (a \rightarrow b) \\ &\quad , Invertible\ [\,a\,], Invertible\ [\,b\,]) \\ &\Rightarrow (a \rightarrow b) \rightarrow [\,b\,] \rightarrow [\,[\,a\,]\,] \end{aligned}$$

In order to provide this unproblematic instance, we first extend the lifted function type with a constructor for functions from the Haskell world. We define the extended lifted function type as a generalized algebraic data type (GADT) [42]. Note that the additional constructor for functions from the Haskell world implicitly uses existential quantification [29, 41] as well as type equality constraints [48].

$$\begin{aligned} & \mathbf{data}\ (\rightarrow_{FL})\ a\ b\ \mathbf{where} \\ &\quad Func_{FL} :: (FL\ a \rightarrow FL\ b) \rightarrow (a \rightarrow_{FL} b) \\ &\quad HaskellFunc :: (Groundable\ (Lifted\ c) \\ &\qquad\qquad , Convertible\ c, Convertible\ d) \\ &\qquad\qquad \Rightarrow (c \rightarrow d) \rightarrow (Lifted\ c \rightarrow_{FL} Lifted\ d) \end{aligned}$$

With the extended data type for lifted functions at hand, we can give the instance of *Convertible* for this type.

$$\begin{aligned}&\textbf{instance}\ (\mathit{Groundable}\ (\mathit{Lifted}\ a), \mathit{Convertible}\ a\\&\qquad\quad , \mathit{Convertible}\ b) \Rightarrow \mathit{Convertible}\ (a \to b)\ \textbf{where}\\&\quad \mathit{to}\ f = f\ \text{`}\mathit{seq}\text{`}\ \mathit{HaskellFunc}\ f\\&\quad \mathit{from}\ (\mathit{HaskellFunc}\ f) = \mathit{unsafeCoerce}\ f\end{aligned}$$

To convert a former Haskell function to the Haskell world again, we just extract the function from the constructor and coerce[12] it to the correct type. It is safe to define *from* partially, because functions cannot appear in the result of an inverse function due to *Invertible* constraints being unsatisfiable for function types.

Finally, we need to adapt our helper function for monadic function application app^{FL} from Section 2.4. When we apply a Haskell function to a lifted value, we first compute the ground normal form of the lifted value before converting and passing it to the function. We accept that, by computing the ground normal form, the application of a Haskell function to a lifted value might be more strict than the author of the function intended.

$$\begin{aligned}&app^{FL} :: FL\ (a \to_{FL} b) \to FL\ a \to FL\ b\\&mf\ \text{`}app^{FL}\text{`}\ x = mf \gg\!=^{FL} \lambda\textbf{case}\\&\quad Func_{FL}\ f \qquad \to f\ x\\&\quad HaskellFunc\ f \to groundFL\ x \gg\!=^{FL} toFL' \circ f \circ from\end{aligned}$$

To account for the fact that partiality of Haskell functions should lead to failed computations instead of run-time errors in the lifted setting (see Section 2.3), we use a modified version of the function *toFL* from Section 4.1 that checks whether its input is partially defined. Testing for partiality is done with *isBottom* from a library[13] by Danielsson and Jansson [11].

$$\begin{aligned}&toFL' :: Convertible\ a \Rightarrow a \to FL\ (Lifted\ a)\\&toFL'\ x = \textbf{if}\ isBottom\ x\ \textbf{then}\ failedFL\ \textbf{else}\ return^{FL}\ (to\ x)\end{aligned}$$

7.2 Primitive Types

We have shown how algebraic data types are lifted and narrowed for our functional logic computations. However, numerical primitive types like *Int* are not actually constructor-based and, thus, our approach is not directly suitable for them. In this section, we outline a constraint-based extension of our functional logic monad to efficiently support primitive types—a topic often neglected in other works. We implemented this extension for our plugin, but omit the implementation details due to a lack of space.

The problematic aspect of primitive data types is best shown in the context of pattern matching. As an example, consider the following function that performs pattern matching on its argument of type *Int*.

$$\begin{aligned}&isZero :: Int \to Bool\\&isZero\ 0 = True\\&isZero\ \_ = False\end{aligned}$$

In the monadically lifted variant, this pattern matching leads to the instantiation of any free variable provided for the argument. Instantiating a free variable of type *Int* would require narrowing all 18 quintillion[14] values for a 64-bit integer and would in turn result in equally many computation branches, which is not remotely feasible in practice.

In order to be more efficient, we introduce constraints whenever pattern matching on a free variable of primitive type occurs. These constraints capture the notion that such a pattern match can either succeed or fail, which corresponds to the variable being equal or unequal to the value matched against. Pattern matching on a free variable of primitive type then only results in two new computation branches instead of the immediate instantiation of the variable. We add an equality constraint in the computation branch where the pattern matching succeeds, whereas we add an inequality constraint in the other branch. This approach is similar to the notion of negative information for pattern match compilation in ML [51] and Haskell [17].

To implement this idea we extend the state of our non-determinism monad from Section 3.1 with a data structure that stores (in-)equality constraints on free variables. We check the consistency of the stored constraints every time we add further constraints. Whenever the constraint store becomes inconsistent at one point of evaluation, we can immediately abandon the corresponding computation branch—potentially without instantiation of an involved free variable at all. We take the constraints stored for a free variable into account to limit the values actually being narrowed when instantiation becomes necessary.

The approach of adding constraints for better support of primitive types is transferable to other primitive types like *Char*, *Word*, etc. If we use finite domain constraints [24], we could even efficiently implement primitive operations like $(+)$ and $(*)$. However, our plugin only uses equality constraints at the moment.

8 Related Work

We identify mainly three fields of related work, which we discuss in the following: the embedding of logic computations in Haskell, other approaches to automatic (partial) inversion, and pattern matching extensions.

8.1 Logic Computations in Haskell

Braßel et al. [8] design a compiler for Curry that targets Haskell. The language is implemented by means of a transformation that—similar to our approach—has to explicitly model

[12] Using *unsafeCoerce* in this context is always safe, because we know that our *Lifted* type family is injective as mentioned in Footnote 8.

[13] Available at https://hackage.haskell.org/package/ChasingBottoms.

[14] 2^{64} = 18,446,744,073,709,551,616 to be exact.

the sharing of non-determinism. Various authors [9, 18] later propose constraint-based extensions of the implementation.

A library for functional logic computations in Haskell by Fischer et al. [15] truly models call-by-need via explicit sharing of monadic computations. They use a state monad with a heap that stores computations instead of values. However, they lack an explicit representation of free variables, which is crucial for our inversion framework. We could integrate their approach to model call-by-need into our effect monad, but at the cost of making the overall framework more complex. For example, higher-kinded polymorphic type variables would require the use of the *QuantifiedConstraints* language extension [22].

Naylor et al. [38] also implement a library for functional logic programming in Haskell. However, they do not support data types with non-deterministic components, which makes their solution not applicable to our setting.

Claessen and Ljunglöf [10] present a library for typed logical variables in Haskell based on an embedding of Prolog [50], but their embedding is strict and, therefore, not suitable for our inversion.

8.2 Automatic Inversion

For this section, we focus on automatic inversion in functional languages, where a lot of work has been done already. In contrast to most other publications, we generally support higher-order functions and seem to be the only ones who address primitive types in the context of inversion.

Romanenko [47] considers the first-order functional programming language Refal. In the process of adding inversion to the language, he ends up with a functional logic extension of Refal, called Refal-R. While his approach has similarities to ours, namely using free variables and narrowing, it is aimed at a strict source and target language that is considerably different from Haskell's semantics and the strongly typed approach considered here.

Abramov and Glück [2] present the Universal Resolving Algorithm (URA) for inverse interpretation in a first-order functional programming language restricted to tail-recursion. Abramov et al. [3] later extend the original URA to general recursion and improve efficiency as well as termination by reducing the search space using lazy evaluation. In our approach, we achieve a similar search space reduction due to Haskell's laziness.

Focusing only on injective functions, where efficient and deterministic inverses are known to exist, Glück and Kawabe [16] present an approach to automatically derive the inverse of first-order functions. They use methods of LR parsing to eliminate non-determinism resulting from their automatic program inversion, which can be done because of their focus on injectivity.

The language Sparcl by Matsuda and Wang [34] aims to incorporate inversion into the language by design. Their approach is similar to reversible languages, but their combination of both invertible and non-invertible computations in one program makes their language less restrictive and allows a seamless integration.

Nishida et al. [39] propose a partial-inversion compiler of constructor term rewriting systems that first generates a conditional term rewriting system and then unravels it to an unconditional system. Their approach is like ours based on narrowing, but only covers strict programming languages.

Almendros-Jiménez and Vidal [4] describe another partial inversion technique—again for first-order functional programs—that is specialized on inductively sequential term rewriting systems. In the same paper, they also sketch extensions of their approach to higher-order and laziness, something our approach also supports.

8.3 Pattern Matching Extensions

Antoy and Hanus [5] introduce functional patterns to Curry, which are implemented using a *functional pattern unification operator* that is similar to our matching function from Section 4.2. The connection between functional patterns and function inversion has also been noted by Braßel and Christiansen [7].

With a similar motivation as functional patterns, *context patterns* as introduced by Mohnen [36] support the definition of functions based on matching subterms at an arbitrary depth. However, they have different semantics from functional patterns and the syntax is less intuitive.

Egi and Nishiwaki [13] present *Egison*, a language with non-linear pattern matching. With a library named *Sweet Egison*[15], Egi et al. [12] offer an embedding of Egison within Haskell. Functional patterns also allow for non-linear pattern matching (see Section 5), a connection we want to further explore in the future.

On a different note, *pattern synonyms* by Pickering et al. [44] allow for a more convenient notation of complex patterns. In contrast to functional patterns, however, functions still cannot be used in patterns (apart from "hiding" them in view patterns). Nevertheless, pattern synonyms integrate nicely with our implementation of functional patterns, because the latter are transformed into regular Haskell patterns.

9 Conclusion

In this paper, we have presented an approach to automatic function inversion in Haskell that covers both inversion and partial inversion using an extension with non-determinism and free variables. Based on inverse functions, we have also shown an implementation of functional patterns—a pattern matching extension previously not available in Haskell. As a proof of concept, we provide our approach as a GHC plugin.

In the future, we plan to prove correctness as well as completeness properties for our automatic function inversion.

[15] Available at https://github.com/egison/sweet-egison.

We furthermore aim to incorporate finite-domain constraints for better support of operations on primitive types in our plugin. Last but not least, we want to examine the implementation of non-linear pattern matching on the basis of functional patterns in more detail.

Acknowledgments

We thank Robert Glück and Michael Hanus for fruitful discussions as well as the anonymous reviewers for their helpful suggestions.

References

[1] Andreas Abel, Marcin Benke, Ana Bove, John Hughes, and Ulf Norell. 2005. Verifying Haskell Programs Using Constructive Type Theory. In *Proceedings of the 2005 ACM SIGPLAN Workshop on Haskell (Haskell '05)*. Association for Computing Machinery, New York, NY, USA, 62–73. https://doi.org/10.1145/1088348.1088355

[2] Sergei Abramov and Robert Glück. 2000. The Universal Resolving Algorithm: Inverse Computation in a Functional Language. In *Mathematics of Program Construction (MPC '00)*. Springer-Verlag, Berlin, Heidelberg, 187–212. https://doi.org/10.1007/10722010_13

[3] Sergei Abramov, Robert Glück, and Yuri Klimov. 2007. An Universal Resolving Algorithm for Inverse Computation of Lazy Languages. In *Proceedings of the 6th International Andrei Ershov Memorial Conference on Perspectives of Systems Informatics (PSI '06)*. Springer-Verlag, Berlin, Heidelberg, 27–40. https://doi.org/10.1007/978-3-540-70881-0_6

[4] Jesús M. Almendros-Jiménez and Germán Vidal. 2007. Automatic Partial Inversion of Inductively Sequential Functions. In *Proceedings of the 18th International Conference on Implementation and Application of Functional Languages (IFL '06)*. Springer-Verlag, Berlin, Heidelberg, 253–270. https://doi.org/10.1007/978-3-540-74130-5_15

[5] Sergio Antoy and Michael Hanus. 2006. Declarative Programming with Function Patterns. In *Proceedings of the 15th International Conference on Logic-Based Program Synthesis and Transformation (LOPSTR '05)*. Springer-Verlag, Berlin, Heidelberg, 6–22. https://doi.org/10.1007/11680093_2

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

[7] Bernd Braßel and Jan Christiansen. 2008. A Relation Algebraic Semantics for a Lazy Functional Logic Language. In *Proceedings of the 10th International Conference on Relational and Kleene Algebra Methods in Computer Science (RelMiCS '08)*. Springer-Verlag, Berlin, Heidelberg, 37–53. https://doi.org/10.1007/978-3-540-78913-0_5

[8] Bernd Braßel, Michael Hanus, Björn Peemöller, and Fabian Reck. 2011. KiCS2: A New Compiler from Curry to Haskell. In *Proceedings of the 20th International Conference on Functional and Constraint Logic Programming (WFLP '11)*, Herbert Kuchen (Ed.). Springer-Verlag, Berlin, Heidelberg, 1–18.

[9] Bernd Braßel, Michael Hanus, Björn Peemöller, and Fabian Reck. 2013. Implementing Equational Constraints in a Functional Language. In *Proceedings of the 15th International Symposium on Practical Aspects of Declarative Languages (PADL '13, Vol. 7752)*, Kostis Sagonas (Ed.). Springer-Verlag, Berlin, Heidelberg, 125–140. https://doi.org/10.1007/978-3-642-45284-0_9

[10] Koen Claessen and Peter Ljunglöf. 2001. Typed Logical Variables in Haskell. *Electronic Notes in Theoretical Computer Science* 41, 1 (2001), 37. https://doi.org/10.1016/S1571-0661(05)80544-4

[11] Nils Anders Danielsson and Patrik Jansson. 2004. Chasing Bottoms. In *International Conference on Mathematics of Program Construction (MPC '04)*. Springer-Verlag, Berlin, Heidelberg, 85–109.

[12] Satoshi Egi, Akira Kawata, Mayuko Kori, and Hiromi Ogawa. 2020. *Sweet Egison: A Haskell Library for Non-Deterministic Pattern Matching.*

[13] Satoshi Egi and Yuichi Nishiwaki. 2018. Non-Linear Pattern Matching with Backtracking for Non-Free Data Types. In *Programming Languages and Systems*, Sukyoung Ryu (Ed.). Vol. 11275. Springer International Publishing, Cham, 3–23. https://doi.org/10.1007/978-3-030-02768-1_1

[14] Martin Erwig and Simon Peyton Jones. 2001. Pattern Guards and Transformational Patterns. *Electronic Notes in Theoretical Computer Science* 41, 1 (2001), 3. https://doi.org/10.1016/S1571-0661(05)80540-7

[15] Sebastian Fischer, Oleg Kiselyov, and Chung-Chieh Shan. 2011. Purely Functional Lazy Nondeterministic Programming. *Journal of Functional Programming* 21, 4-5 (Sept. 2011), 413–465. https://doi.org/10.1017/S0956796811000189

[16] Robert Glück and Masahiko Kawabe. 2004. Derivation of Deterministic Inverse Programs Based on LR Parsing. In *Proceedings of the 7th International Conference on Functional and Logic Programming (FLOPS '04)*. Springer-Verlag, Berlin, Heidelberg, 291–306. https://doi.org/10.1007/978-3-540-24754-8_21

[17] Sebastian Graf, Simon Peyton Jones, and Ryan G. Scott. 2020. Lower Your Guards: A Compositional Pattern-Match Coverage Checker. *Proceedings of the ACM on Programming Languages* 4, ICFP, Article 107 (Aug. 2020). https://doi.org/10.1145/3408989

[18] Michael Hanus, Björn Peemöller, and Jan Rasmus Tikovsky. 2014. Integration of Finite Domain Constraints in KiCS2. In *Proceedings of the 7th Working Conference on Programming Languages (ATPS '14, Vol. 1129)*. CEUR, 151–170.

[19] Michael Hanus and Finn Teegen. 2020. Adding Data to Curry. In *Declarative Programming and Knowledge Management (WFLP '19)*. Springer International Publishing, Cham, 230–246. https://doi.org/10.1007/978-3-030-46714-2_15

[20] Michael Hanus (ed.). 2016. *Curry: An Integrated Functional Logic Language (Version 0.9.0)*.

[21] M. C.B. Hennessy and E. A. Ashcroft. 1977. Parameter-Passing Mechanisms and Nondeterminism. In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing (STOC '77)*. Association for Computing Machinery, New York, NY, USA, 306–311. https://doi.org/10.1145/800105.803420

[22] Ralf Hinze and Simon Peyton Jones. 2000. Derivable Type Classes. In *Proceedings of the 2000 Haskell Workshop (Haskell '00)*.

[23] John Hughes. 1999. Restricted Data Types in Haskell. In *Proceedings of the 1999 Haskell Workshop (Haskell '99)*.

[24] J. Jaffar and J.-L. Lassez. 1987. Constraint Logic Programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '87)*. Association for Computing Machinery, New York, NY, USA, 111–119. https://doi.org/10.1145/41625.41635

[25] Johan Jeuring, Patrik Jansson, and Cláudio Amaral. 2012. Testing Type Class Laws. In *Proceedings of the 2012 Haskell Symposium (Haskell '12)*. Association for Computing Machinery, New York, NY, USA, 49–60. https://doi.org/10.1145/2364506.2364514

[26] Mark P. Jones. 1993. 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)*. Association for Computing Machinery, New York, NY, USA, 52–61. https://doi.org/10.1145/165180.165190

[27] Mark P. Jones. 1995. Functional Programming with Overloading and Higher-Order Polymorphism. In *Advanced Functional Programming*, Gerhard Goos, Juris Hartmanis, Jan Leeuwen, Johan Jeuring, and Erik Meijer (Eds.). Vol. 925. Springer-Verlag, Berlin, Heidelberg, 97–136. https://doi.org/10.1007/3-540-59451-5_4

[28] Andrew J. Kennedy and Dimitrios Vytiniotis. 2012. Every Bit Counts: The Binary Representation of Typed Data and Programs. *Journal of Functional Programming* 22, 4–5 (Aug. 2012), 529–573. https://doi.org/

10.1017/S0956796812000263

[29] Konstantin Läufer and Martin Odersky. 1994. Polymorphic Type Inference and Abstract Data Types. *ACM Transactions on Programming Languages and Systems* 16, 5 (Sept. 1994), 1411–1430. https://doi.org/10.1145/186025.186031

[30] Sheng Liang, Paul Hudak, and Mark Jones. 1995. Monad Transformers and Modular Interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '95)*. Association for Computing Machinery, New York, NY, USA, 333–343. https://doi.org/10.1145/199448.199528

[31] Geoffrey Mainland. 2007. Why It's Nice to Be Quoted: Quasiquoting for Haskell. In *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop (Haskell '07)*. Association for Computing Machinery, New York, NY, USA, 73–82. https://doi.org/10.1145/1291201.1291211

[32] Simon Marlow (ed.). 2010. *Haskell 2010 Language Report*.

[33] Kazutaka Matsuda and Meng Wang. 2013. FliPpr: A Prettier Invertible Printing System. In *Proceedings of the 22nd European Conference on Programming Languages and Systems (ESOP '13)*. Springer-Verlag, Berlin, Heidelberg, 101–120. https://doi.org/10.1007/978-3-642-37036-6_6

[34] Kazutaka Matsuda and Meng Wang. 2020. Sparcl: A Language for Partially-Invertible Computation. *Proc. ACM Program. Lang.* 4, ICFP (Aug. 2020). https://doi.org/10.1145/3409000

[35] John McCarthy. 1956. The Inversion of Functions Defined by Turing Machines. In *Automata Studies, Annals of Mathematical Studies*, J. McCarthy C.E. Shannon (Ed.). Number 34. Princeton University Press, 177–181.

[36] Markus Mohnen. 1996. Context Patterns in Haskell. In *Selected Papers from the 8th International Workshop on Implementation of Functional Languages (IFL '96)*. Springer-Verlag, Berlin, Heidelberg, 41–57.

[37] Kazutaka Morita, Akimasa Morihata, Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. 2007. Automatic Inversion Generates Divide-and-Conquer Parallel Programs *(PLDI '07)*. Association for Computing Machinery, New York, NY, USA, 146–155. https://doi.org/10.1145/1250734.1250752

[38] Matthew Naylor, Emil Axelsson, and Colin Runciman. 2007. A Functional-Logic Library for Wired. In *Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop (Haskell '07)*. Association for Computing Machinery, New York, NY, USA, 37–48. https://doi.org/10.1145/1291201.1291207

[39] Naoki Nishida, Masahiko Sakai, and Toshiki Sakabe. 2005. Partial Inversion of Constructor Term Rewriting Systems. In *Proceedings of the 16th International Conference on Term Rewriting and Applications (RTA '05)*. Springer-Verlag, Berlin, Heidelberg, 264–278. https://doi.org/10.1007/978-3-540-32033-3_20

[40] Dominic Orchard and Tom Schrijvers. 2010. Haskell Type Constraints Unleashed. In *Proceedings of the 10th International Conference on Functional and Logic Programming (FLOPS '10)*. Springer-Verlag, Berlin, Heidelberg, 56–71. https://doi.org/10.1007/978-3-642-12251-4_6

[41] Nigel Perry. 1991. *The Implementation of Practical Functional Programming Languages*. PhD Thesis. University of London.

[42] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. 2006. Simple Unification-Based Type Inference for GADTs. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming (ICFP '06)*. Association for Computing Machinery, New York, NY, USA, 50–61. https://doi.org/10.1145/1159803.1159811

[43] Simon Peyton Jones, Stephanie Weirich, Richard A. Eisenberg, and Dimitrios Vytiniotis. 2016. A Reflection on Types. In *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, Sam Lindley, Conor McBride, Phil Trinder, and Don Sannella (Eds.). Springer International Publishing, Cham, 292–317. https://doi.org/10.1007/978-3-319-30936-1_16

[44] Matthew Pickering, Gergő Érdi, Simon Peyton Jones, and Richard A. Eisenberg. 2016. Pattern Synonyms. In *Proceedings of the 9th International Symposium on Haskell (Haskell '16)*. Association for Computing Machinery, New York, NY, USA, 80–91. https://doi.org/10.1145/2976002.2976013

[45] Uday S. Reddy. 1985. Narrowing as the Operational Semantics of Functional Languages. In *Proceedings of the 1985 Symposium on Logic Programming (SLP '85)*. IEEE-CS, Boston, Massachusetts, USA, 138–151.

[46] Tillmann Rendel and Klaus Ostermann. 2010. Invertible Syntax Descriptions: Unifying Parsing and Pretty Printing. In *Proceedings of the Third ACM Haskell Symposium on Haskell (Haskell '10)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/1863523.1863525

[47] Alexander Romanenko. 1991. Inversion and Metacomputation. In *Proceedings of the 1991 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '91)*. Association for Computing Machinery, New York, NY, USA, 12–22. https://doi.org/10.1145/115865.115868

[48] Tom Schrijvers, Simon Peyton Jones, Manuel Chakravarty, and Martin Sulzmann. 2008. Type Checking with Open Type Functions. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP '08)*. Association for Computing Machinery, New York, NY, USA, 51–62. https://doi.org/10.1145/1411204.1411215

[49] Neil Sculthorpe, Jan Bracker, George Giorgidze, and Andy Gill. 2013. The Constrained-Monad Problem. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13)*. Association for Computing Machinery, New York, NY, USA, 287–298. https://doi.org/10.1145/2500365.2500602

[50] Silvija Seres and Michael Spivey. 1999. Embedding Prolog in Haskell. In *Proceedings of the 1999 Haskell Workshop (Haskell '99)*.

[51] Peter Sestoft. 1996. ML Pattern Match Compilation and Partial Evaluation. In *Partial Evaluation*, Olivier Danvy, Robert Glück, and Peter Thiemann (Eds.). Springer-Verlag, Berlin, Heidelberg, 446–464. https://doi.org/10.1007/3-540-61580-6_22

[52] Tim Sheard and Simon Peyton Jones. 2002. Template Meta-Programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell (Haskell '02)*. Association for Computing Machinery, New York, NY, USA, 1–16. https://doi.org/10.1145/581690.581691

[53] Saurabh Srivastava, Sumit Gulwani, Swarat Chaudhuri, and Jeffrey S. Foster. 2011. Path-Based Inductive Synthesis for Program Inversion. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. Association for Computing Machinery, New York, NY, USA, 492–503. https://doi.org/10.1145/1993498.1993557

[54] Jan Stolarek, Simon Peyton Jones, and Richard A. Eisenberg. 2015. Injective Type Families for Haskell. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell (Haskell '15)*. Association for Computing Machinery, New York, NY, USA, 118–128. https://doi.org/10.1145/2804302.2804314

[55] Philip Wadler. 1985. How to Replace Failure by a List of Successes: A Method for Exception Handling, Backtracking, and Pattern Matching in Lazy Functional Languages. In *Proceedings of a Conference on Functional Programming Languages and Computer Architecture (FPCA '85)*. Springer-Verlag, Berlin, Heidelberg, 113–128.

[56] Philip Wadler. 1987. Efficient Compilation of Pattern-Matching. In *The Implementation of Functional Programming Languages*, Simon L. Peyton Jones (Ed.). Prentice-Hall, 78–103. https://doi.org/10.1016/0141-9331(87)90510-2

[57] Philip Wadler. 1987. 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)*. Association for Computing Machinery, New York, NY, USA, 307–313. https://doi.org/10.1145/41625.41653

[58] Philip Wadler. 1990. Comprehending Monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming (LFP '90)*. Association for Computing Machinery, New York, NY, USA, 61–78. https://doi.org/10.1145/91556.91592

[59] Philip Wadler and Stephen Blott. 1989. 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)*. Association for Computing Machinery, New York, NY, USA, 60–76. https://doi.org/10.1145/75277.75283

[60] Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. 2012. Giving Haskell a Promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI '12)*. Association for Computing Machinery, New York, NY, USA, 53–66. https://doi.org/10.1145/2103786.2103795