Functional Logic Program Transformations

Michael Hanus[1] and Steven Libby[2]

[1] Institut für Informatik, Kiel University, Kiel, Germany
mh@informatik.uni-kiel.de
[2] University of Portland, Portland, Oregon, U.S.A.
libbys@up.edu

Abstract. Many tools used to process programs, like compilers, analyzers, or verifiers, perform transformations on their intermediate program representation, like abstract syntax trees. Implementing such program transformations is a non-trivial task, since it is necessary to iterate over the complete syntax tree and apply various transformations at nodes in a tree. In this paper we show how the features of functional logic programming are useful to implement program transformations in a compact and comprehensible manner. For this purpose, we propose to write program transformations as partially defined and non-deterministic operations. Since the implementation of non-determinism usually causes some overhead compared to deterministically defined operations, we compare our approach to a deterministic transformation method. We evaluate these alternatives for the functional logic language Curry and its intermediate representation FlatCurry which is used in various analysis and verification tools and compilers.

1 Introduction

Program transformation is a very old idea in computer science, dating all the way back to McCarthy [22]. It has proved to be an important tool in program analysis, verification, and compilation. Functional and logic languages in particular have seen many examples of harnessing program transformation to perform complex tasks. Peyton Jones demonstrated how a Haskell compiler could be constructed as a sequence of program transformations [27] and how optimizations could be implemented as more program transformations. [25]. Appel provided several examples of program transformations to construct an ML compiler [8] including transforming the AST into CPS form and several optimizations. Flanagan et al's original formulation of A-normal form was given as a set of rewrite rules [10], and Peyton Jones et al. showed how general rewriting can be useful for easily implementing optimizations [26]. Many optimizations are presented as simple program transformations. For example, Gill's shortcut deforestation [11], inlining and beta reduction [24], as well as more complex optimizations like partial evaluation [19].

Functional logic languages such as Curry[3] also take advantage of program transformations. The RICE compiler is implemented as a series of program trans-

[3] https://www.curry-lang.org/

formations [21]. Hanus also showed how program transformations can be used to simplify the verification of Curry programs [14]. Peemöller's partial evaluator also uses several program transformations [16].

Although the theory of transformations is usually presented as a simple and elegant way to work with complex structures, the reality is usually much more complex. For example, the A-normal from transformation [10] is given as a set of three rewrite rules with an evaluation context. However, the actual implementation is a half a page of Scheme code using a moderately complex function to build up a continuation to perform the rewrite. While it is remarkable that the implementation is only a half page, it has lost the readability of the original rewrite rules. This is a reoccurring theme in program analysis and compilers. The program transformations in the GHC compiler are often long inscrutable functions that bear little resemblance to the transformation they implement.

In this paper, we show how features from functional logic programming can be used to represent transformations in a modular, composable way. The sections that follow review functional logic programming and Curry, describe its intermediate representation called FlatCurry, introduce our approach to defining transformations, examine their results, and conclude.

2 Functional Logic Programming with Curry

The declarative language Curry [18] supports features from functional programming (demand-driven evaluation, strong typing, higher-order functions) as well as from logic programming (computing with partial information, unification, constraints), see [5,13] for surveys. The syntax of Curry is close to Haskell [23]. In addition to Haskell, Curry applies rules with overlapping left-hand sides in a (don't know) non-deterministic manner (where Haskell always selects the first matching rule) and allows *free* (*logic*) *variables* in conditions and right-hand sides of defining rules. The operational semantics is based on demand-driven evaluation which is optimal for large classes of programs [3].

Similarly to Haskell, Curry is strongly typed so that a program consists of data type definitions defining the *constructors* of these types and *functions* or *operations* on these types. The following simple example defines the concatenation operation "`++`" on lists and an operation `adjDup` which returns an adjacent duplicate number in a list:[4]

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

adjDup :: [Int] → Int
adjDup xs | xs =:= _ ++ [x,x] ++ _
          = x    where x free
```

`adjDup` exploits the unification operator "`=:=`" which instantiates free variables when both expressions are evaluated and unified. `adjDup` is also called a *non-deterministic operation* since it might deliver more than one result for a fixed

[4] To check some unintended errors, Curry requires the explicit declaration of free variables, as `x` in the rule of `adjDup`. This is not necessary for anonymous variables which are denoted by an underscore.

argument, e.g., `adjDup [1,2,2,1,3,3,4]` yields `2` and `3`. Non-deterministic operations, which are interpreted as mappings from values into sets of values [12], are an important feature of contemporary functional logic languages. One particularly important operation is the choice operator "`?`" which non-deterministically returns one of its two arguments. As we will see, such operations are useful to specify program transformations in a compact manner.

The operation `adjDup` is reducible if the actual argument has the form as specified in the right-hand side of the condition's equation. For such cases, Curry supports a more compact notation:

```
adjDup' :: [Int]  → Int
adjDup' (_ ++ [x,x] ++ _) = x
```

Since the pattern used in `adjDup'` contains the defined function "`++`", it is called a *functional pattern*. A functional pattern denotes all standard patterns to which the functional pattern can be evaluated. Functional pattern matching can be efficiently implemented by a specific unification procedure [4]. Functional patterns can express pattern matching at arbitrary depths so that they are useful to specify program transformations.

Functional patterns can be supported in Curry due to its logic programming features, i.e., the ability to deal with non-deterministic and failing computations. To control non-deterministic computations and failures, Curry supports *encapsulated search operators* which return all or some values of an expression. In this paper, we use only the search operator

```
oneValue :: a  → Maybe a
```

which returns `Nothing` if the argument has no value, otherwise `Just` some value.[5]

3 FlatCurry: An Intermediate Represention for Curry Programs

Curry has many more features than described in the previous section, like type classes, monadic I/O, modules, etc. To avoid the consideration of all these features in language processing tools for Curry (compilers, analyzers,...), such tools often use an intermediate language where the syntactic sugar of the source language has been eliminated and the pattern matching strategy is explicit. This intermediate language is called FlatCurry and will be the basis of example program transformations described in this paper so that we describe it in more detail. Apart from compilers, FlatCurry has been used to specify the operational semantics of Curry programs [1], to implement a modular framework for the analysis of Curry programs [17], or to verify non-failing properties of Curry programs [14].

[5] The precise selection of the value is unspecified so that the operation is considered unsafe (there are also other more complex declarative encapsulation operators in Curry). Since we are not interested in the confluence of program transformations, this slightly non-declarative behavior is acceptable.

$$
\begin{array}{lll}
P & ::= & D_1 \ldots D_m & \text{(program)} \\
D & ::= & f(x_1, \ldots, x_n) = e & \text{(function definition)} \\
e & ::= & x & \text{(variable)} \\
 & | & f(e_1, \ldots, e_n) & \text{(function/constructor application)} \\
 & | & e_1 \ \mathit{or}\ e_2 & \text{(disjunction)} \\
 & | & \mathit{let}\ x_1, \ldots, x_n\ \mathit{free}\ \mathit{in}\ e & \text{(free variables)} \\
 & | & \mathit{let}\ x = e\ \mathit{in}\ e' & \text{(let binding)} \\
 & | & \mathit{case}\ e\ \mathit{of}\ \{p_1 \to e_1; \ldots; p_n \to e_n\} & \text{(case expression)} \\
p & ::= & c(x_1, \ldots, x_n) & \text{(pattern)}
\end{array}
$$

Fig. 1. Syntax of the intermediate language FlatCurry

Figure 1 summarizes the abstract syntax of FlatCurry. A FlatCurry program consists of a sequence of function definitions (we omit data type definitions here), where each function is defined by a single rule. Patterns in source programs are compiled into case expressions and overlapping rules are joined by explicit disjunctions. The patterns in each case expression are required to be non-overlapping.

Any Curry program can be transformed into this format [2,7]. In particular, the front end of a Curry compiler transforms source programs into FlatCurry programs so that FlatCurry is the intermediate language of many Curry compilers. Therefore, it is a reasonable target for program transformations that simplify or optimize Curry programs.

For instance, consider the following operation to insert an element at an arbitrary position into a list:

```
insert :: a  →  [a]  →  [a]
insert x ys     = x : ys
insert x (y:ys) = y : insert x ys
```

This definition, which has overlapping rules, can be transformed into the FlatCurry definition (where we use the standard list notation)

```
insert(x,xs) =    x : xs
               or case xs of { y:ys  →  y : insert(x,ys) }
```

In order to process FlatCurry programs inside Curry programs, there is a Curry package `flatcurry`[6] defining data types to represent FlatCurry programs and operations to read Curry programs and returning the equivalent FlatCurry program as terms of these data types. To understand the transformation examples discussed later, we show the data types to represent FlatCurry expressions. In order to distinguish the different kinds of applications, the following type is used:

```
data CombType = FuncCall | ConsCall
              | FuncPartCall Int | ConsPartCall Int
```

[6] `https://cpm.curry-lang.org/pkgs/flatcurry.html`

Hence, `FuncCall` and `ConsCall` are used in applications of functions and constructors, respectively, whereas `FuncPartCall` and `ConsPartCall` are used in partial applications where the integer argument specifies the number of missing arguments. Then FlatCurry expressions are represented by the following types (for the sake of readability, this definition is slightly simplified compared to the actual `flatcurry` package):

```
data Expr
  = Var  Int
  | Comb CombType String [Expr]
  | Or   Expr Expr
  | Free [Int] Expr
  | Let  (Int, Expr) Expr
  | Case Expr [BranchExpr]

data BranchExpr = Branch Pattern Expr

data Pattern = Pattern String [Int]
```

Note that variables are represented as unique integers. For instance, consider the Boolean negation operation `not`. Its FlatCurry definition is

```
not(x) = case x of { False  → True ; True  → False }
```

Its right-hand side expression is represented by the following data term (where variable x has the index `0`)::

```
Case (Var 0)
     [Branch (Pattern "False" []) (Comb ConsCall "True" []),
      Branch (Pattern "True"  []) (Comb ConsCall "False" [])]
```

The prelude operation "`$`", defined by

```
($) :: (a  → b)  → a  → b
f $ x = f x
```

is an infix application operator often used to write applications without parentheses. The Curry expression "`not $ not True`" is represented as the following data term:

```
Comb FuncCall "$" [Comb (FuncPartCall 1) "not" [],
                   Comb FuncCall "not" [Comb ConsCall "True []]]
```

In the next section we discuss a program transformation to simplify this expression by removing the call to "`$`".

4 Transformations on FlatCurry Programs

Now that we have a structure to transform Curry programs, we can define methods to transform them.

4.1 Functional logic transformations

In order to simplify the development of program transformations, we would like a simple, composable way to represent a single transformation. With this repre-

```
orFloat (Or (Let vs e1) e2) = Let vs (Or e1 e2)
orFloat (Or e1 (Let vs e2)) = Let vs (Or e1 e2)
```

Fig. 2. Transformation: float a let expression out of a choice expression

```
unDollar (dollar f args miss x)
 | miss == 1 = Comb FuncCall f (args++[x])
 | miss > 1  = Comb (FuncPartCall (miss-1)) f (args++[x])

dollar f args miss x = Comb FuncCall "$"
                            [Comb (FuncPartCall miss) f args, x]
```

Fig. 3. Transformaton: remove a call to "`$`" in an expression

sentation, we can separate the logic for traversing a FlatCurry expression from the transformation itself. The most natural option is to represent a transformation as a function on FlatCurry expressions. We allow our transformations to be both non-deterministic and partial. For instance, we want to implement a transformation which moves a local let binding out of a choice. This can be expressed by the following transformation rules:

$$\begin{array}{rcl} (let\ x = e\ in\ e_1)\ ?\ e_2 & \Rightarrow & let\ x = e\ in\ (e_1\ ?\ e_2) \\ e_1\ ?\ (let\ x = e\ in\ e_2) & \Rightarrow & let\ x = e\ in\ (e_1\ ?\ e_2) \end{array}$$

This transformation is correct under the assumption that local variables always have distinct identifiers (which is ensured by the Curry front end). The immediate implementation in Curry is shown in Figure 2. Note that `orFloat` fails on non-matching arguments and is overlapping on the expression

```
(let x = 1 in x) ? (let y = 1 in y)
```

The non-determinism is not an issue here because both rules will apply eventually and the order of application does not matter. This allows us to avoid encoding unnecessary control flow information when it is not important.

Non-determinism also allows us to use functional patterns in transformation rules. The undollar example in Figure 3 implements the transformation

$$f\ \$\ x\ \Rightarrow\ f\ x$$

The operation `dollar` abbreviates a FlatCurry expression which is an application of "`$`" where the first argument is a partially applied function (and not a variable or another expression). The use of `dollar` as a pattern in `unDollar` shows how functional patterns can be used to improve the readability of transformations. While we could have specified the entire pattern for the application of the "`$`" function in the definition of `unDollar`, the functional pattern makes the transformation clearer.

We also allow transformations to be partial functions. This means that a transformation may not apply in all cases. This is not an issue. If the transformation fails to apply, we ignore it and move to the next one. This is particularly

```
caseCancel (Case (Comb ConsCall c []) (withBranch c e)) = e
withBranch c e = (_ ++ [Branch (Pattern c []) e] ++ _)
```

Fig. 4. Case canceling transformation for constructors with no arguments.

helpful when we need to search for a specific subexpression such as in the case canceling example in Figure 4 which implements the following transformation:

$$case\ C\ of\ \{\ldots; C \to e; \ldots\} \ \Rightarrow\ e$$

We use a functional pattern to find the specific branch that contains the value of the scrutinee of the case. Requiring this function to be total would necessitate searching through the cases manually until we find one or signal a failure if one is not found.

A more subtle use of partial transformations occurs in the undollar example in Figure 3. If the function `f` is a partial application with `0` arguments missing, then we should not turn it into a partial call expecting `-1` arguments. By omitting this case, we have neatly sidestepped this issue.

So far, our transformations have the type `Expr` → `Expr`. While this is sufficient for simple transformations, it is helpful to augment this type with extra information used by more complex transformations. Thus, we add a further argument consisting of the index of the next fresh variable available for use and the path to the current subexpression we are transforming (expressed by the type `Path` which is a type synonym for a list of integers). This allows us to generate new variables as needed, and determine useful information such as whether we are at the root of right-hand side expression.

In addition to the transformed expression, a transformation also returns the number of fresh variables used in the transformed expression. This allows us to keep track of the next fresh variable efficiently. Thus, the full type of a general expression transformation is

```
type ExprTransformation = (Int, Path) → Expr → (Expr,Int)
```

We can lift a simple transformation of type `Expr` → `Expr` to `ExprTransformation` with the function

```
makeT :: (Expr → Expr) → ExprTransformation
makeT f = \_ e → (f e, 0)
```

Finally, our transformation library provides an operation (its implementation is discussed in Sect. 4.3)

```
transformExpr :: (() → ExprTransformation) → Expr → Expr
```

which applies a transformation repeatedly until no more transformations can be applied. To avoid committing to a specific choice of a non-deterministic transformation too early, the transformation is not passed as a constant but as a function which takes a dummy unit argument of type `()`.

Individual transformations such as `unDollar` or `orFloat` can be useful, but the real power of this system comes from composability. We support two types of

composition. Serial composition is applying one transform after another, while parallel composition applies both transforms at the same time. In our approach, applying transformations in series is just function composition: for instance, applying first transformation `t1` and afterwards transformation `t2` can be expressed in Curry with the function composition operator "`.`" by

```
transformExpr (\() → t2) . transformExpr (\() → t1)
```

The parallel composition of `t1` and `t2`, i.e., applying both transformations `t1` and `t2` whenever possible, can be expressed by

```
transformExpr (\() → t1 ? t2)
```

This composition is expressive and gives us control over how we execute our transformations. For example, we may want to float all of the let expressions out of a choice before we remove all "`$`" applications and cancel simple cases. This can be expressed by

```
transformExpr (\() → makeT unDollar ? makeT caseCancel) .
  transformExpr (\() → makeT orFloat)
```

4.2 Purely functional transformations

Separation of concerns and composability are powerful tools in our system. However, we may wish to avoid all non-determinism in a program transformation. Although this sacrifices convenience, we can still support composition with total deterministic functions.

Deterministic transformations must be totally defined so that they have type `Expr → Maybe Expr`. A successful transformation returns `Just e`, where a failing transformation returns `Nothing`.

The `caseCancelDET` example below shows the changes we need to make to define a deterministic transformation. Functional patterns are replaced with total case expressions. However, because we need to find the branch in a list, we need to write an auxiliary function to find the correct branch. The case expressions ensure that pattern matching is done in sequence so we do not have overlapping patterns.

```
caseCancelDET e = case e of
                    Case (Comb ConsCall c []) bs → find c bs
                    _                            → Nothing
 where find _ [] = Nothing
       find c (Branch (Pattern p vs) e : bs)
         | c == p && null vs = Just e
         | otherwise         = find n bs
```

To apply a deterministic transformation repeatedly to all subexpressions until no more transformation is possible, our library provides an operation

```
transformExprDet :: ExprTransformation → Expr → Expr
```

Note that the unit argument used in `transformExpr` is not necessary since the transformation argument is always deterministic.

Sequential composition is unchanged from the previous version, but we need a new operator for parallel composition. The `<?>` operator tries one transformation, and falls back on the second if the first transformation fails.

```
t1 <?> t2 = \env e  → case t1 env e of
                        Nothing  → t2 env e
                        answer   → answer
```

With this new composition operator, we can define a deterministic version of our previous transformation with the following (where we omit the definition of the deterministic versions of `unDollear` and `orFloat`):

```
transformExprDet (makeT unDollarDet <?> makeT caseCancelDet) .
  transformExprDet (makeT orFloatDet)
```

4.3 Transformation strategies

In order to apply the transformation rules shown so far to arbitrary subexpressions of an expression (typically, the right-hand side of a function definition), we need a transformation engine. This is the job of the operation `transformExpr`. To evaluate our framework, we use three different strategies for applying transformations which are sketched in the following.

The first is the *chaotic strategy* (*CS*). This strategy non-deterministically selects a subexpression, tries to apply a transformation on this subexpression, and, if possible, replaces it by the result of the transformation This is similar to the deep selection pattern [6]. When applying a transformation rule to the selected subexpression, the operation `oneValue` is used to check whether a transformation is applicable and to ensure that only one replacement is performed. This is the easiest way to implement but might be inefficient for large programs.[7]

The second strategy is the *deterministic strategy* (*DS*). This strategy traverses the expression in a bottom-up fashion, while attempting to apply a purely functional transformation at each subexpression. If it applies, the subexpression is replaced and we try again, otherwise we move on.

The final strategy is a combination of the two that we call the *mixed strategy* (`MS`). This strategy still traverses the syntax tree in a bottom up manner, but non-deterministic transformation rules are applied at each step where `oneValue` is used to force the result to be deterministic. This improves on the chaotic strategy because we do not need to recompute the path every time a transformation applies.

With a framework for creating, composing, and applying transformations, we turn our attention to evaluating the effectiveness of the different strategies. We assess both the performance as well as the development cycle for transformations.

[7] An implementation of this strategy in Curry is shown in Appendix A.

				PAKCS			KiCS2		
Module	Size	Funcs	Trans	CS	MS	DS	CS	MS	DS
Prelude	72485	1285	3	955	1494	321	795	306	148
Data.Char	2190	9	0	22	49	7	18	7	4
Data.Either	1693	11	0	3	3	1	1	1	1
Data.List	14841	87	0	54	65	20	24	18	9
Data.Maybe	1809	9	0	4	6	1	3	2	1
Numeric	3494	7	4	9	18	4	17	17	2
System.Console.GetOpt	17328	47	0	66	119	21	38	23	11
System.IO	6223	51	0	15	19	7	5	5	3

Table 1. Transforming standard libraries with the transformations of Sect. 4.1

5 Benchmarks

We have presented different methods to implement transformations in a declarative language. Functional logic transformations exploit partially defined nondeterministic operations to specify transformations in a comprehensible manner and avoid superfluous code for non-matching cases. Their actual implementation demands for logic programming features like controlling failures and nondeterminism via encapsulated search. In contrast, purely functional transformations require the encoding of all cases in a deterministic manner but their implementation does not require logic programming features. Since the use of logic features requires some additional efforts at run time, it is interesting to know about the price to pay for supporting compact and comprehensible transformation specifications. Therefore, we evaluate the approaches discussed above in this section.

We implemented the chaotic, mixed, and deterministic transformation strategies in Curry so that the transformation rules can be written as shown above. In particular, transformations written in a functional logic style can be used for both the chaotic and mixed strategy, whereas the deterministic strategy requires the more complex style of deterministic transformations in the form of totally defined operations.

In the first set of benchmarks, we applied all three transformation rules shown in Sect. 4.1 to various standard libraries contained in Curry distributions. For each module, Table 1 shows the size of the Curry source file, the number of defined functions in the corresponding FlatCurry program (note that the transformations are applied to the right-hand side of each function), the number of transformations performed in the module, and the time (in milliseconds) to apply the various strategies described in Sect. 4.3.[8] Furthermore, we executed the benchmarks with the Curry implementations PAKCS [15], which compiles into Prolog, and KiCS2 [9], which compiles into Haskell. Since the standard libraries evolved over years so that superfluous pieces of code are avoided, there are only

[8] We measured the transformation times on a Linux machine running Ubuntu 22.04 with an Intel Core i7-1165G7 (2.80GHz) processor with eight cores.

Module	Size	Funcs	Trans	PAKCS CS	PAKCS MS	PAKCS DS	KiCS2 CS	KiCS2 MS	KiCS2 DS
Prelude	72485	1285	5779	39339	77830	3042	15660	5272	2740
Data.Char	2190	9	163	1358	3189	90	365	86	76
Data.Either	1693	11	12	6	10	2	2	3	3
Data.List	14841	87	237	481	870	70	117	61	56
Data.Maybe	1809	9	29	103	175	13	39	11	11
Numeric	3494	7	47	54	93	9	10	8	6
System.Console.GetOpt	17328	47	418	1483	2566	139	331	126	108
System.IO	6223	51	89	50	89	15	9	11	9

Table 2. Transforming standard libraries to A-normal form

a few transformations which can be applied. Thus, this benchmark mainly evaluates the time to try to apply transformations at all positions in a program w.r.t. different strategies.

As one can expect, the deterministic transformation strategy is the fastest. However, the times with the mixed strategy are not so much worse. Thus, there is no need to rewrite functional logic transformations into the more complex purely deterministic style, in particular, if an efficient Curry implementation like KiCS2 is used. It is also interesting that the chaotic strategy is faster than the mixed strategy when it is executed with PAKCS.. The reason is unclear but it should be noted that Prolog has a direct support for non-determinism whereas KiCS2 explicitly implements non-determinism via search tree structures.

In order to get an impression for the transformation behavior when many transformations can actually be applied, we tested our implementation with a transformation of FlatCurry programs into their A-Normal Form (ANF) [10]. ANF requires that operations or constructors are only applied to variables. ANF is used in many compilers and also in the specification of operational semantics for functional [20] and functional logic [1] programs. The ANF transformation replaces non-variable arguments by fresh variables which are introduced in let bindings. For instance, the A-Normal Form of the operation `insert` defined in Sect. 3 is

```
insert(x,xs) =    x : xs
              or case xs of { y:ys  → let z = insert(x,ys)
                                    in y : z }
```

The functional logic ANF transformation basically consists of a single rule which replaces a non-variable argument by a new variable and the corresponding let binding. On the other hand, the purely functional ANF transformation has to implement the same transformation but must also consider all other expressions in order to avoid a failure.

The results of applying the ANF transformation to the set of standard libraries are shown in Table 2. The run times indicate that the differences between the non-deterministic and the deterministic transformation can get larger when many transformations are applicable, which is the case for the Prolog-

based PAKCS compiler. However, for the more efficient Haskell-based KiCS2 compiler, the differences are not so high, in particular, when the mixed strategy is used instead of the chaotic strategy. Since the same transformation rules can be used for both strategies, the mixed strategy can always be preferred. Taking into account that functional logic transformations are easier to implement, the differences in the absolute timings are acceptable when an efficient Curry system, like KiCS2, is used.

6 Conclusions

In this paper we presented methods to implement program transformations in a declarative manner. Our examples target the intermediate language FlatCurry but similar transformations can be written for other languages if their abstract syntax trees are represented as data terms. Although such transformations can be implemented in any programming language, we showed that the functional logic programming features of Curry are useful to write compact and comprehensible transformations so that the implemented code is quite similar to specifications of such transformations. The definition of transformations as partially defined and non-deterministic operations avoids writing superfluous code to control the transformations.

Qualitatively, our system for program transformations offers several advantages to writing the transformations by hand. One of the most significant benefits is that the transformations are written entirely in Curry. There is no new language to learn or tool to install. Instead, each transformation is a Curry function. This allows Curry developers to write complex transformations without needing to learn a new system. Developers with experience in functional languages can typically understand a transformation with limited explanation.

Another advantage is that transformations are extensible. Because transformations are Curry functions, we can make them as complex as necessary. Transformations are not restricted to being simple rewrite rules, but can involve arbitrary computations. Transformations can be extended to handle additional information. For example, a beta-reduction transformation can take a map containing information about previously seen functions. We simply add another parameter to the transformation.

A third advantage is that our system can easily be extended to report which transformations are applied. By supplying a name to each transformation, it becomes easy to reconstruct the entire transformation derivation. This is an important tool for debugging transformations. It enables developers to see how expressions evolved from the original form to the final result, which is often difficult when working with several transformations.

Our system has been used extensively in the RICE compiler [21] to implement complex transformations including conversion to A-normal form, variable inlining, and beta-reduction. Development of these optimizations was usually straightforward, and we were able to see how the optimizations interacted with

one another. It also made it easier to modify the optimizations to make them more effective.

Our system provides a convenient, concise method of specifying program transformations entirely in Curry. By allowing both partial and non-deterministic operations, we are able to focus on the transformations themselves. This system is extensible and has been proven effective in large programs like the RICE compiler.

References

1. E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. Operational semantics for declarative multi-paradigm languages. *Journal of Symbolic Computation*, 40(1):795–829, 2005.
2. S. Antoy. Constructor-based conditional narrowing. In *Proc. of the 3rd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2001)*, pages 199–206. ACM Press, 2001.
3. S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *Journal of the ACM*, 47(4):776–822, 2000.
4. S. Antoy and M. Hanus. Declarative programming with function patterns. In *Proceedings of the International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'05)*, pages 6–22. Springer LNCS 3901, 2005.
5. S. Antoy and M. Hanus. Functional logic programming. *Communications of the ACM*, 53(4):74–85, 2010.
6. S. Antoy and M. Hanus. New functional logic design patterns. In *20th International Workshop on Functional and (constraint) Logic Programming (WFLP 2011)*, pages 19–34, Odense, Denmark, 2011. Springer LNCS 6816.
7. S. Antoy, M. Hanus, A. Jost, and S. Libby. ICurry. In *Declarative Programming and Knowledge Management - Conference on Declarative Programming (DECLARE 2019)*, pages 286–307. Springer LNCS 12057, 2020.
8. A. Appel. *Compiling with Continuations*. Cambridge University Press, New York, NY, USA, 2007.
9. B. Braßel, M. Hanus, B. Peemöller, and F. Reck. KiCS2: A new compiler from Curry to Haskell. In *Proc. of the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011)*, pages 1–18. Springer LNCS 6816, 2011.
10. C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The Essence of Compiling with Continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, pages 237–247, 1993.
11. A. Gill, J. Launchbury, and S. L. Peyton Jones. A Short Cut to Deforestation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, FPCA '93, pages 223–232, New York, NY, USA, 1993.
12. J.C. González-Moreno, M.T. Hortalá-González, F.J. López-Fraguas, and M. Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming*, 40:47–87, 1999.
13. M. Hanus. Functional logic programming: From theory to Curry. In *Programming Logics - Essays in Memory of Harald Ganzinger*, pages 123–168. Springer LNCS 7797, 2013.

14. M. Hanus. Inferring non-failure conditions for declarative programs. In *Proc. of the 17th International Symposium on Functional and Logic Programming (FLOPS 2024)*, pages 167–187. Springer LNCS 14659, 2024.
15. M. Hanus, S. Antoy, B. Braßel, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, F. Steiner, and F. Teegen. PAKCS: The Portland Aachen Kiel Curry System. Available at `https://www.curry-lang.org/pakcs/`, 2025.
16. M. Hanus and B. Peemöller. A partial evaluator for Curry. In *Proc. of the 23rd International Workshop on Functional and (Constraint) Logic Programming (WFLP 2014)*, pages 55–71. Universität Halle-Wittenberg, 2014.
17. M. Hanus and F. Skrlac. A modular and generic analysis server system for functional logic programs. In *Proc. of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation (PEPM'14)*, pages 181–188. ACM Press, 2014.
18. M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.9.0). Available at `http://www.curry-lang.org`, 2016.
19. N. Jones, C. Gomard, and P. Sestoft. *Partial evaluation and automatic program generation*. Prentice-Hall, Inc., USA, 1993.
20. J. Launchbury. A natural semantics for lazy evaluation. In *Proc. 20th ACM Symposium on Principles of Programming Languages (POPL'93)*, pages 144–154. ACM Press, 1993.
21. S. Libby. RICE: An optimizing Curry compiler. In *25th International Symposium Practical Aspects of Declarative Languages (PADL 2023)*, pages 3–19. Springer LNCS 13880, 2023.
22. J McCarthy. A basis for a mathematical theory of computation, preliminary report. In *Papers Presented at the May 9-11, 1961, Western Joint IRE-AIEE-ACM Computer Conference*, page 225–238, New York, NY, USA, 1961. Association for Computing Machinery.
23. S. Peyton Jones, editor. *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press, 2003.
24. S. Peyton Jones and S. Marlow. Secrets of the Glasgow Haskell Compiler Inliner. *J. Funct. Program.*, 12(5):393–434, July 2002.
25. S. Peyton Jones and A. Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32(1), October 1997.
26. S. Peyton Jones, A. Tolmach, and T. Hoare. Playing by the rules: Rewriting as a practical optimisation technique in GHC. *Haskell 2001*, 04 2001.
27. S.L. Peyton Jones. Compiling Haskell by program transformation: A report from the trenches. In *Proc. ESOP'96*, pages 18–44. Springer LNCS 1058, 1996.

A Implementation of the Chaotic Strategy

In the following we show the implementation of the chaotic transformation strategy in Curry. First, we define the non-deterministic operation `subExpOf` which returns, for a given FlatCurry expression, some subexpression and its path (a list of integers representing argument positions counted from zero).

```
subExpOf :: Expr → (Path,Expr)
subExpOf e = ([],e) -- the subexpression is the entire expression
subExpOf (Comb _ _ args) =
  uncurry extendPath $ anyOf (zip [0..] (map subExpOf args))
subExpOf (Let (_,e) _) = extendPath 0 (subExpOf e)
subExpOf (Let _     e) = extendPath 1 (subExpOf e)
subExpOf (Free _ e) = extendPath 1 (subExpOf e)
subExpOf (Or e1 e2) =
  extendPath 0 (subExpOf e1) ? extendPath 1 (subExpOf e2)
subExpOf (Case ce _) = extendPath 0 (subExpOf ce)
subExpOf (Case _ bs) = uncurry extendPath $
  anyOf (zip [1..] (map (subExpOf . branchExp) bs))
 where branchExp (Branch _ be) = be
```

The auxiliary operation `extendPath` extends the path component by one position:

```
extendPath :: Int → (Path,Expr) → (Path,Expr)
extendPath pos (p,e) = (pos:p, e)
```

Now we can define the chaotic strategy as the operation `transformExpr` which simplifies a FlatCurry expression by iteratively applying the given transformation rule to some subexpression. Thus, a transformation step is implemented by selecting a subexpression with `subExpOf` where a rule can be applied (implemented by the local operation `tryTransExpr`). To make the overall strategy deterministic and non-failing, we control the transformation with `oneValue`. The operation `newVar` returns the next fresh variable not occurring in an expression, and the operation `replace` replaces a subexpression in an expression at the given position (its definition is a similar case distinction as in `subExpOf`).

```
transformExpr :: (() → ExprTransformation) → Expr → Expr
transformExpr trans n e = runTrExpr trans (newVar e) e

runTrExpr :: (() → ExprTransformation) → Int → Expr → Expr
runTrExpr trans nvar exp =
  case oneValue (tryTransExpr nvar exp) of
    Nothing             → exp -- no transformation applicable
    Just (p, (e',nvs))  → runTrExpr trans (nvar+nvs)
                                     (replace exp p e')
 where
  tryTransExpr v e = let (p,se) = subExpOf e
                     in (p, trans () (v,p) se)
```