Transforming Functional Logic Programs into Monadic Functional Programs

Bernd Braßel Sebastian Fischer Michael Hanus Fabian Reck

Institut für Informatik, CAU Kiel, D-24098 Kiel, Germany
{bbr|sebf|mh|fre}@informatik.uni-kiel.de

Abstract. We present a high-level transformation scheme to translate lazy functional logic programs into pure Haskell programs. This transformation is based on a recent proposal to efficiently implement lazy non-deterministic computations in Haskell into monadic style. We build on this work and define a systematic method to transform lazy functional logic programs into monadic programs with explicit sharing. This results in a transformation scheme which produces high-level and flexible target code. For instance, the target code is parametric w.r.t. the concrete evaluation monad. Thus, different monad instances could, for example, define different search strategies (e.g., depth-first, breadth-first, parallel). We formally describe the basic compilation scheme and some useful extensions.

1 Introduction

Functional logic languages (see [10] for a recent survey) integrate the most important features of functional and logic languages to provide a variety of programming concepts to the programmer. In particular, modern languages of this kind, such as Curry [13] or $\mathcal{TOY}$ [16], amalgamate the concepts of demand-driven evaluation from functional programming with non-deterministic evaluation from logic programming. This combination is not only desirable to obtain efficient evaluation but it has also a positive effect on the search space, i.e., lazy evaluation on non-deterministic programs yields smaller search spaces due to its demand-driven exploration of the search space (compare [10]).

Although the combination of such features is quite useful for high-level application programming, their implementation is challenging. Many older implementations (e.g., [11, 16, 18]) are based on Prolog's depth-first backtracking strategy to explore the search space. Since this strategy leads to operational incompleteness and reduces the potential of modern architectures for parallelism, more recent implementations of functional logic languages offer more flexible search strategies (e.g., [5, 7]). In order to avoid separate implementations for different strategies, it would be desirable to specify the search strategy (e.g., depth-first, breadth-first, parallel) as a parameter of the implementation. A first step towards such an implementation has been done in [8] where a Haskell library for non-deterministic programming w.r.t. different strategies is proposed. In this paper, we use this idea to compile functional logic programs into pure Haskell programs that are parameterized such that the generated code works with different run-time systems (various search strategies, call-time/run-time choice, etc).

Before presenting our compilation scheme, we review the features of functional logic programming that we are going to implement as well as the language Curry that we use for concrete examples. From a syntactic point of view, a Curry program is a functional program (with a Haskell-like syntax [19]) extended by non-deterministic rules and free (logic) variables in defining rules. For the sake of simplicity, we do not consider free variables, since it has been shown that they can be replaced by non-deterministic rules [4]. Actually, we use a kernel language, also called *overlapping inductively sequential programs* [2], which are functional programs extended by a (don't know) non-deterministic choice operator "`?`". This is not a loss of generality, since (1) any functional logic program (with extra variables and conditional, overlapping rules) can be transformed into an overlapping inductively sequential program [1], and (2) narrowing computations in inductively sequential programs with free variables are equivalent to computations in overlapping inductively sequential programs without free variables [4, Th. 2].

A *functional logic program* consists of the definition of functions and data types on which the functions operate. For instance, the data types of Booleans and polymorphic lists are defined as

```
data Bool   = True | False
data List a = Nil  | Cons a (List a)
```

Concatenation of lists and an operation that duplicates a list can be defined as:

```
append :: List a  → List a  → List a
append Nil         ys = ys
append (Cons x xs) ys = Cons x (append xs ys)

dup :: List a  → List a
dup xs = append xs xs
```

Note that functional logic programs require a strict separation between *constructor symbols* (like `True`, `False`, `Nil`, or `Cons`) and *defined functions* or *operations* (like `append` or `dup`). In contrast to general term rewriting, the formal parameters in a rule defining a function contain only variables and constructor symbols. This restriction also holds for pure functional or logic programs and is important to provide efficient evaluation strategies (see [10] for more details).

Logic programming aspects become relevant when considering *non-deterministic operations*, i.e., operations that yield more than one result. For this purpose, there is a distinguished choice operator "`?`" which returns non-deterministically one of its arguments. For instance, the following operation returns a one-element list containing either `True` or `False`:

```
oneBool :: List Bool
oneBool = Cons (True ? False) Nil
```

Now, consider an expression that duplicates the result of the previous operation:

```
main = dup oneBool
```

What are possible values of `main`? One could argue (in pure term rewriting) that "`Cons True (Cons False Nil)`" is a value of `main` by deriving it to "`append oneBool oneBool`", and then the first argument to "`Cons True Nil`" and the second to "`Cons False Nil`" (this semantics is called *run-time choice* [14]). But this result is not desired as the operation `dup` is intended to *duplicate* a given list (rather than return the concatenation of two different lists). In or-

der to obtain this behavior, González-Moreno et al. [9] proposed a rewriting logic as a logical (execution- and strategy-independent) foundation for declarative programming with non-strict and non-deterministic operations. This logic specifies the *call-time choice* semantics [14] where values of the arguments of an operation are determined before the operation is evaluated. Note that this does not necessarily mean that operations are evaluated eagerly. One can still evaluate operations lazily provided that actual arguments passed to operations are shared. For instance, the two occurrences of argument "`xs`" of operation `dup` are shared, i.e., the actual argument `oneBool` is evaluated to the same value on both positions. Thus, "`Cons True (Cons True Nil)`" and "`Cons False (Cons False Nil)`" are the only values of `main`, as intended. Detailed descriptions of this operational semantics can be found in [9, 10].

In functional logic programs, non-deterministic operations can occur in any level of the program, in particular, inside nested structures, as shown in operation `oneBool` above. This makes the transformation of such programs into pure functional programs non-trivial. For instance, the traditional functional representation of non-deterministic computations as "lists of successes" [20] is not easily applicable, as one might expect, due to the arbitrary nesting of non-deterministic operations. In the following section we review a recent solution to this problem [8].

2 Lazy, Monadic Non-determinism

In the previous section, we have introduced Curry which combines demand driven with non-deterministic evaluation. While both features can be easily expressed separately in a functional language, their combination is non-trivial. In this section we summarize previous work [8] that shows why.

Demand-driven evaluation is built into *lazy* execution mechanisms of the functional language Haskell. Laziness combines *non-strict* execution (expressions are evaluated only if needed) with *sharing* (expressions are evaluated at most once). Non-deterministic evaluation can be simulated in Haskell via lists or, more generally, non-determinism monads, i.e., instances of the `MonadPlus` type class. The `MonadPlus` type class specifies the following overloaded operations to express non-deterministic computations.[1]

```
mzero   :: MonadPlus m ⇒ m a
return  :: MonadPlus m ⇒ a → m a
mplus   :: MonadPlus m ⇒ m a → m a → m a
(>>=)   :: MonadPlus m ⇒ m a → (a → m b) → m b
```

`mzero` denotes a failing computation, i.e., one without results, `return` creates a deterministic computation, i.e., one with a single result, `mplus` creates a non-deterministic choice between the results of the two argument computations, and "`>>=`" applies a non-deterministic function to every result of a non-deterministic computation. For lists, `mzero` is the empty list, `return` creates a singleton list, `mplus` is list concatenation, and `>>=` (pronounced 'bind') can be implemented by mapping the given function over the given list and concatenating the results.

[1] In fact, `return` and "`>>=`" have more general types because they are not only available in non-determinism monads but in arbitrary instances of the `Monad` type class.

The Curry expression `(True ? False)` can be expressed monadically:

```
trueOrFalse :: MonadPlus ⇒ m Bool
trueOrFalse = mplus (return True) (return False)
```

The constructors `True` and `False` are wrapped with `return` and the resulting computations are combined with `mplus` which replaces Curry's non-deterministic choice operator "`?`". When evaluated in the list monad, `trueOrFalse` yields `[True,False]` which can be verified in a Haskell environment:

```
ghci> trueOrFalse :: [Bool]
[True,False]
```

However, different implementations of the `MonadPlus` interface can be used, e.g., to influence the search strategy. If we use the `Maybe` monad rather than the list monad, we just get one result in depth-first order:

```
ghci> trueOrFalse :: Maybe Bool
Just True
```

The overloading of `trueOrFalse` allows us to execute it using different types. Programs that are compiled with our transformation scheme are also overloaded and can be executed by different monad instances.

We motivate the monadic implementation that we use in our transformation by a sequence of ideas that leads to the final design. A simple idea to translate the Curry operation `oneBool` into monadic Haskell is to reuse the existing Curry data types and *bind* non-deterministic arguments of their constructors:

```
oneBoolM1 :: MonadPlus m ⇒ m (List Bool)
oneBoolM1 = trueOrFalse >>= λb → return (Cons b Nil)
```

We feed the result of function `trueOrFalse` above into a singleton list using the "`>>=`" operator. Like the corresponding Curry operation, `oneBoolM1` yields a singleton list that contains either `True` or `False` non-deterministically:

```
ghci> oneBoolM1 :: [List Bool]
[Cons True Nil, Cons False Nil]
```

However, there is a subtle difference w.r.t. laziness. In Curry, `oneBool` yields the head-normal form of its result without executing the non-deterministic choice inside the list, whereas `oneBoolM1` first executes the non-deterministic choice between `True` and `False` and yields a list with a deterministic first element in each non-deterministic branch of the computation. Whereas in Curry, non-determinism can be nested inside data structures, the monadic non-determinism presented so far cannot.

To overcome this limitation, we can use data types with nested non-deterministic components. Nested monadic lists can be defined by wrapping each constructor argument with an additional type parameter "`m`" that represents a non-determinism monad:

```
data MList m a = MNil | MCons (m a) (m (MList m a))
```

The additional "`m`"s around the arguments of `MCons` allow to wrap non-deterministic computations inside lists. Here is a different translation of the Curry operation `oneBool` into monadic Haskell:

```
oneBoolM :: MonadPlus m ⇒ m (MList m Bool)
oneBoolM = return (MCons trueOrFalse (return MNil))
```

This function *deterministically* yields a singleton list with an element that is a non-deterministic choice:

```
ghci> oneBoolM :: [MList [] Bool]
[MCons [True,False] [MNil]]
```

This translation of the Curry operation is more accurate w.r.t. laziness because the `MCons` constructor can be matched without distributing the non-determinism in its first argument. In order to print such nested non-deterministic data in the usual way, we need to distribute non-determinism to the top level [8].

Now that we have changed the list data type in order to support nested non-determinism, we need to re-implement the list functions defined in Section 1. The monadic variant of the `dup` function takes a monadic list as argument and yields a monadic list as result:

```
dupM1 :: MonadPlus m ⇒ m (MList m a) → m (MList m a)
dupM1 xs = appendM xs xs
```

Similarly, the monadic variant of `append` takes two monadic lists and yields one.

```
appendM :: MonadPlus m ⇒ m (MList m a) → m (MList m a)
                       → m (MList m a)
appendM l ys =
 l >>= λl' → case l' of
               MNil → ys
               MCons x xs → return (MCons x (appendM xs ys))
```

This definition resembles the Curry definition of `append` but additionally handles the monadic parts inside and around lists. In order to match on the first argument "`l`" of `appendM`, we *bind* one of its non-deterministic head-normal forms to the variable "`l'`". Depending on the value of "`l'`", `appendM` yields either the second argument "`ys`" or a list that contains the first element "`x`" of "`l'`" and the result of a recursive call (which can both be non-deterministic).

Although such a translation with nested monadic data accurately models non-strictness, it does not ensure sharing of deterministic results. The definition of `dupM1` given above uses the argument list "`xs`" twice and hence, the value of "`xs`" is shared via Haskell's built-in laziness. However, in `dupM1` the variable "`xs`" denotes a *non-deterministic computation* that yields a list and the built-in sharing does not ensure that both occurrences of "`xs`" in `dupM1` denote the same *deterministic result* of this computation. Hence, the presented encoding of nested monadic data implements run-time choice instead of call-time choice:

```
ghci> dupM1 oneBoolM :: [MList [] Bool]
[MCons [True,False] [MCons [True,False] [MNil]]]
```

When distributed to the top-level, the non-determinism in the list elements leads to lists with different elements because the information that both elements originate from the same expression is lost.

The conflict between non-strictness and sharing in presence of monadic non-determinism has been resolved recently using an additional monadic combinator for explicit sharing [8]:

```
share :: (Sharing m, Shareable m a) ⇒ m a → m (m a)
```

The type class context of `share` specifies that "`m`" (referring to a non-determinism monad throughout this paper) and the type denoted by "`a`" support explicit sharing. Using `share`, the Curry function `dup` can be translated as follows:

```
dupM :: (MonadPlus m, Sharing m, Shareable m a) ⇒
        m (MList m a)  → m (MList m a)
dupM xs = share xs  >>=  λxs  →  appendM xs xs
```

The result of `share xs` is a monadic computation that yields itself a monadic computation which is similar to "`xs`" but denotes the same deterministic result when used repeatedly. Hence, the argument "`xs`" to `appendM` (which intentionally shadows the original argument "`xs`" of `dupM`) denotes the same deterministic list in both argument positions of `appendM` which ensures call-time choice. When executing "`dupM oneBoolM`" in a non-determinism monad with explicit sharing, the resulting lists do not contain different elements.

The library that implements `share`, and that we use to execute transformed functional logic programs, is available online[2]. The implementation ideas, the operation that allows to observe results of computations with explicit sharing, as well as equational laws that allow to reason about such computations are not in the scope of this paper but described elsewhere [8].

3 Transforming Functional Logic Programs

In this section we formally define the transformation of functional logic programs into monadic functional programs, i.e., pure Haskell programs. In order to simplify the transformation scheme, we consider functional logic programs in flat form as a starting point of our transformation. Flat programs are a standard representation for functional logic programs where the strategy of pattern matching is explicitly represented by case expressions. Since source programs can be easily translated into the flat form [12], we omit further details about the transformation of source programs into flat programs but define the syntax of flat programs before we present our transformation scheme.

3.1 Syntax of Flat Functional Logic Programs

As a first step we fix the language of polymorphic type expressions. We denote by $\overline{o_n}$ the sequence of objects $o_1, \ldots, o_n$.

Definition 1 (Syntax of Type Expressions). *Type expressions are either type variables α or type constructors T applied to type expressions:*

$$\tau ::= \alpha \mid T(\overline{\tau_n})$$

Function types are of the form $\overline{\tau_n} \to \tau$ where $\overline{\tau_n}, \tau$ are type expressions. We denote by $\mathcal{T}$ the set of all function types.

As discussed in Section 1, functional logic programs contain program rules as well as declarations of data types. We summarize type declarations in the notion of a program signature.

[2] http://sebfisch.github.com/explicit-sharing

Definition 2 (Program signature). *A program signature is a pair (Σ, ty) where $\Sigma = \mathcal{F} \uplus \mathcal{C}$ is the disjoint union of a set $\mathcal{F}$ of function symbols and a set $\mathcal{C}$ of constructor symbols. The mapping $ty : \Sigma \to \mathcal{T}$ maps each symbol in Σ to a function type such that, for all $C \in \mathcal{C}$, $ty(C) = \overline{\tau} \to T(\overline{\alpha})$ for a type constructor T. If $ty(s) = \overline{\tau_n} \to \tau$, then n is called the* arity *of symbol s, denoted by $ar(s)$.*

The signature for the program of Section 1 contains the following symbols

$$\mathcal{C} = \{True, False, Nil, Cons\} \qquad \mathcal{F} = \{append, dup, oneBool, main\}$$

as well as the following mapping of types:

$$\begin{aligned} ty(Nil) &= \to List\ a \\ ty(Cons) &= a, List\ a \to List\ a \\ &\vdots \\ ty(append) &= List\ a, List\ a \to List\ a \\ ty(dup) &= List\ a \to List\ a \\ &\vdots \end{aligned}$$

Next we fix the syntax of programs w.r.t. a given program signature. We consider flat programs where pattern matching is represented by case expressions.

Definition 3 (Syntax of Programs). *Let (Σ, ty) be a program signature specifying the types of all constructor and functions symbols occurring in a program and $\mathcal{X}$ be a set of variables disjoint from the symbols occurring in Σ. A* pattern *is a constructor $C \in \mathcal{C}$ applied to pairwise different variables $\overline{x_n}$ where $n = ar(C)$:*

$$p ::= C(\overline{x_n})$$

Expressions *over (Σ, ty) are variables, constructor or function applications, case expressions, or non-deterministic choices:*

$$\begin{array}{lll} e ::= x & & x \in \mathcal{X} \text{ is a variable} \\ \quad | \ C(\overline{e_n}) & & C \in \mathcal{C} \text{ is an } n\text{-ary constructor symbol} \\ \quad | \ f(\overline{e_n}) & & f \in \mathcal{F} \text{ is an } n\text{-ary function symbol} \\ \quad | \ case\ e\ of\ \{\overline{p_n \to e_n}\} & & p_i \text{ have pairwise different constructors} \\ \quad | \ e_1 \,?\, e_2 & & \end{array}$$

Programs *over (Σ, ty) contain for each n-ary function symbol $f \in \mathcal{F}$ one rule of the form $f(\overline{x_n}) \to e$ where $\overline{x_n}$ are pairwise different variables and e is an expression.*

The rules corresponding to the functions `append` and `oneBool` of Section 1 are:

$$\begin{aligned} append(xs, ys) &\to \text{case } xs\ of\ \{\ Nil \to ys, \\ &\qquad\qquad Cons(z, zs) \to Cons(z, append(zs, ys))\ \} \\ oneBool &\to Cons(True \,?\, False, Nil) \end{aligned}$$

For simplicity, we assume that expressions and programs are well typed w.r.t. the standard Hindley/Milner type system. Furthermore, we assume that there is no shadowing of pattern variables, i.e., the variables occurring in the patterns of a case expression are fresh in the scope of the case expression.

Note that all constructor and function symbols are fully applied. The extension to higher-order functions is discussed separately in Section 4.

3.2 Transforming Data Types

In the following transformations, we assume that `m` is a new type variable that does not occur in the program to be transformed. This type variable will denote the monad that implements non-deterministic evaluations in the target program. Since evaluations can be non-deterministic in all levels of functional logic programs, we have to insert `m` as a new argument in all data types. Thus, we start the definition of our transformation by stating how type expressions of functional logic programs are mapped to Haskell type expressions, adding `m` to all argument types.

Definition 4 (Transforming Types). *The transformation $tr(\tau)$ on type expressions τ is defined as follows:*

$$\begin{aligned} tr(\overline{\tau_n} \to \tau) &= \texttt{m}\ tr(\tau_1) \to \ldots \to \texttt{m}\ tr(\tau_n) \to \texttt{m}\ tr(\tau) \\ tr(\alpha) &= \alpha \\ tr(T(\overline{\tau_n})) &= \texttt{(}T\ \texttt{m}\ tr(\tau_1) \ldots tr(\tau_n)\texttt{)} \end{aligned}$$

The transformation of data type declarations adds `m` to all constructors:

Definition 5 (Transforming Data Declarations). *For each type constructor T of arity n, let $\{\overline{C_k}\} = \{\ C \in \mathcal{C} \mid ty(C) = \cdots \to T(\overline{\alpha_n})\ \}$ be the set of constructor symbols for this type constructor. Then we transform the definition of type constructor T into the following Haskell data type declaration:*

$$\begin{aligned} &\texttt{data}\ T\ \texttt{(m :: } * \to * \texttt{)}\ \alpha_1\ \ldots\ \alpha_n\ \texttt{=}\ C_1\ \texttt{(m}\ tr(\tau_{11})\texttt{)}\ \ldots\ \texttt{(m}\ tr(\tau_{1n_1})\texttt{)} \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid\ \ldots \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid\ C_k\ \texttt{(m}\ tr(\tau_{k1})\texttt{)}\ \ldots\ \texttt{(m}\ tr(\tau_{kn_k})\texttt{)} \end{aligned}$$

where $ty(C_j) = \overline{\tau_{jn_j}} \to T(\overline{\alpha_n})$.

The kind annotation `(m :: * → *)` in the previous definition is necessary for data types which have 0-ary data constructors only (i.e., enumeration types). Without this annotation, a wrong kind for `m` would be deduced in this case due to default assumptions in the Haskell type inferencer. Hence, for data types with at least one non-constant data constructor, the kind annotation can be omitted. For instance, the data types presented in the example of Section 1 are transformed into the following Haskell data type declarations:

```
data Bool (m :: * → *) = True | False
data List m a = Nil | Cons (m a) (m (List m a))
```

3.3 Transforming Functions

As discussed in Section 2, variables that have multiple occurrences in the body of a program rule have to be shared in order to conform to the intended call-time choice semantics of functional logic programs. In order to introduce sharing for such variables in our transformation, we need the notion of the number of free occurrences of a variable in an expression:

Definition 6 (Free Occurrences of a Variable). *The number of free occurrences of variable x in expression e, denoted by $occ_x(e)$, is defined as:*

$$occ_x(y) = \begin{cases} 1, & \textit{if } x = y \\ 0, & \textit{otherwise} \end{cases}$$

$$occ_x(s(\overline{e_n})) = \textstyle\sum_{i=1}^{n} occ_x(e_i)$$

$$occ_x(e_1 \,?\, e_2) = max\{occ_x(e_1), occ_x(e_2)\}$$

$$occ_x(\textit{case } e \textit{ of } \{\overline{p_n \to e_n}\}) = \begin{cases} 0, & \textit{if } x \textit{ occurs in some } p_i \ (1 \le i \le n) \\ occ_x(e) + max\{occ_x(e_i) \mid 1 \le i \le n\}, & \textit{otherwise} \end{cases}$$

By $vars_n(e)$ we denote the set of variables occurring at least n times in e:

$$vars_n(e) = \{x \in \mathcal{X} \mid occ_x(e) \ge n\}$$

Note that we count multiple occurrences for each possible computation path. Thus, the variable occurrences in the two branches of a non-deterministic choice expression are not added but only the maximum is considered, i.e., if a variable occurs only once in each alternative of a choice, it is not necessary to share it. The same is true for the branches of a case expression.

In order to translate functional logic expressions into Haskell, we have to apply two basic transformations: (1) Introduce sharing for all variables with multiple occurrences (defined by the transformation *sh* below) and (2) Translate non-deterministic into monadic computations (defined by the transformation *tr* below). Note that these transformations are mutually recursive.

Definition 7 (Transforming Expressions). *The transformation $sh(e)$ introduces sharing for all variables with multiple occurrences in the expression e:*

$$sh(e) = share(vars_2(e), tr(e))$$

$$share(\{\overline{x_k}\}, e) = \begin{cases} \texttt{share}\ x_1\ \texttt{>>=}\lambda x_1\ \to \\ \vdots \\ \texttt{share}\ x_k\ \texttt{>>=}\lambda x_k\ \to \\ e \end{cases}$$

For the sake of simplicity, we do not rename the variables when introducing sharing but exploit the scoping of Haskell, i.e., the argument x_i of `share` *is different from the argument x_i in the corresponding lambda abstraction.*

Transformation tr replaces non-deterministic choices by monadic operations and introduces sharing for the pattern variables of case expressions, if necessary:

$$\begin{aligned} tr(x) &= x \\ tr(f(\overline{e_n})) &= \texttt{(}f\ tr(e_1)\ \ldots\ tr(e_n)\texttt{)} \\ tr(C(\overline{e_n})) &= \texttt{(return (}C\ tr(e_1)\ \ldots\ tr(e_n)\texttt{))} \\ tr(e_1 \,?\, e_2) &= \texttt{(mplus}\ tr(e_1)\ tr(e_2)\texttt{)} \end{aligned}$$

$$tr(\textit{case } e \textit{ of } \{\overline{p_n \to e_n}\}) = \begin{cases} \texttt{(}tr(e)\ \texttt{>>=}\lambda\texttt{x}\ \to \\ \texttt{case x of} \\ \quad p_1\ \to sh(e_1) \\ \quad \vdots \qquad\qquad\qquad\qquad \textit{where } \texttt{x} \textit{ fresh} \\ \quad p_n\ \to sh(e_n) \\ \quad \_\ \to \texttt{mzero)} \end{cases}$$

Note that patterns of case expressions p_i must also be translated into their curried form in Haskell, i.e., each pattern $p_i = C(\overline{x_k})$ is translated into $C\ x_1 \ldots x_k$, but we omit this detail in the definition of tr for the sake of simplicity.

Now we are ready to describe the transformation of program rules by transforming the rule's right-hand side. In addition, we have to add the necessary class dependencies in the type of the defined function as discussed in Section 2.

Definition 8 (Transforming Program Rules). *Let (Σ, ty) be a program signature and $f(\overline{x_n}) \to e$ a rule of a functional logic program. We transform this rule into the following Haskell definition, where $\alpha_1, \ldots, \alpha_k$ are all type variables occurring in $ty(f)$:*

```
f :: (MonadPlus m, Sharing m,
      Shareable m α1, ..., Shareable m αk) ⇒ tr(ty(f))
f x1 ... xn = sh(e)
```

According to the transformation scheme, the rules corresponding to operations `append`, `dup`, and `oneBool` (cf. Section 1) are translated to the Haskell definitions:

```
append :: (MonadPlus m, Sharing m, Shareable m a) ⇒
          m (List m a) → m (List m a) → m (List m a)
append xs ys = xs >>= λ l →
  case l of Nil        → ys
            Cons z zs  → return (Cons z (append zs ys))

dup :: (MonadPlus m, Sharing m, Shareable m a) ⇒
       m (List m a) → m (List m a)
dup xs = share xs >>= λ xs → append xs xs

oneBool :: (MonadPlus m, Sharing m) ⇒ m (List m (Bool m))
oneBool = return (Cons (mplus (return True) (return False))
                       (return Nil))
```

4 Extensions

Up to now, we have described a basic transformation of a first-order kernel language. In this section, we discuss extensions of this transformation scheme.

4.1 Higher-Order Programs

Higher-order programs can be translated with an extension of our transformation scheme. We omit some details like the transformation of higher-order function and data types due to lack of space.

In functional (logic) languages, functions are first class citizens which means that functions can have other functions both as argument and as result. In order to add higher-order features to our source language, we extend it with lambda abstractions and higher-order applications:

$$e ::= \cdots \mid \lambda x \to e \mid apply(e_1, e_2)$$

We still require applications of function and constructor symbols to respect the arity of the corresponding symbol. Over-applications can be expressed using *apply* and partial applications can be transformed into applications of lambda abstractions. For example, the partial application "`append oneBool`" in Curry would be expressed as

$$apply(\lambda xs \to \lambda ys \to append(xs, ys), oneBool)$$

in our source language. Note that we do not use the simpler representation

$$\lambda ys \to append(oneBool, ys)$$

which has a different semantics in Curry because *oneBool* would not be shared if this lambda abstraction is duplicated.

We use the function `iterate` as an example for a higher-order function:

```
iterate :: (a → a) → a → List a
iterate f x = Cons x (iterate f (f x))
```

The function `iterate` yields a list of iterated applications of a given function to a value. In the definition, both arguments of `iterate` are shared. Therefore, the transformation scheme of Section 3.3 would introduce sharing as follows:

```
iterate :: (MonadPlus m, Sharing m, Shareable m a) ⇒
           m (m a → m a) → m a → m (List m a)
iterate f x = share f >>= λ f →
              share x >>= λ x →
              return (Cons x (iterate f (apply f x)))
```

The `apply` function is used to transform the higher-order application of the variable "`f`" to "`x`" and is implemented as follows:

```
apply :: MonadPlus m ⇒ m (m a → m b) → m a → m b
apply f x = f >>= λf → f x
```

In order to translate the Curry expression "`iterate (append oneBool) Nil`", we transform the partial application of append as illustrated above and then apply the following rule to transform lambda abstractions:

$tr(\lambda x \to e) =$ `return (λ x →share x >>= λx →` $tr(e)$`)`

Note, that we precautionary share every argument of a lambda abstraction regardless of whether it is shared in the body or not. This is necessary because the lambda abstraction itself could be duplicated and the argument must be shared also if only duplicated indirectly along with the lambda abstraction. We cannot share already supplied arguments when partial applications are duplicated because we reuse Haskell's higher-order features and, hence, partial Curry applications are represented as Haskell functions that we cannot inspect.

Here is the transformed version of the above call to `iterate`:

```
iterate (apply (return (λ xs → share xs >>= λ xs →
                return (λ ys → share ys >>= λ ys →
                append xs ys)))
               oneBool)
        (return Nil)
```

This translation shares the result of `oneBool` just like the original Curry expression when the argument "`f`" of `iterate` is duplicated. Therefore, the result of this call is an infinite list of boolean lists of increasing length where all elements are either `True` or `False` but no list contains both `True` and `False`.

4.2 An Optimized Transformation Scheme

In this section we present a technique to optimize the programs obtained from the transformation in Section 3.3. The basic idea is that the original transformation scheme may introduce sharing too early. To keep things simple, Definitions 7 and 8 introduce sharing at the beginning of a rule or the case branches, respectively. This scheme is straightforward and a similar scheme is used in PAKCS [11]. When implementing the transformation presented here, we observed that sharing could also be introduced individually for each variable as "late" as possible. Consequently, the ideas presented in this section could also be employed to improve existing compilers like that of PAKCS.

What does "late sharing" mean? Reconsider the transformed `iterate` function given in Section 4.1. Due to the nature of `iterate`, the result is a potentially infinite list. Therefore, in any terminating program the context of a call to iterate will only demand the `x` of the result but not the value of the expression `iterate f (f x)`. It is clear that for yielding `x` in this case there is no need to share `f` (again). Thus, sharing `f` later will improve the resulting code:

```
iterate f x = share x >>= λ x → 
              return (Cons x (share f >>= λ f → 
                                 iterate f (apply f x)))
```

The example also shows that the optimization requires to introduce sharing individually for each variable.

How can we obtain this optimization in general? The idea is that the transformation of expressions needs some information about which variables occur in its context. Whenever the situation arises that for a term $s(\overline{e_n})$ a variable occurs in more than one of the e_n but not in the context, we have to introduce sharing for x right around the result of transforming $s(\overline{e_n})$. Therefore, the transformation tr is extended by an additional argument indicating the set of variables occurring in the context. These ideas are formalized in the following definitions.

First we formalize the idea that variables "occur in more than one" of a sequence of given expressions.

Definition 9. $multocc(\overline{e_n}) = \{x \mid \exists i \neq j : x \in vars_1(e_i) \cap vars_1(e_j)\}$

The optimizing transformation scheme for expressions is then expressed in the following definition. There, the transformation gets as an additional argument the set of variables for which sharing was already introduced. For a variable that does not occur in that set, sharing will be introduced in two situations: (a) before an application if it occurs in more than one argument, or (b) before a case expression "case e *of* $\{\overline{p_n \to e_n}\}$" if it occurs in e *and in at least one* of the branches $\overline{e_n}$.

Definition 10 (Optimized Transformation of Expressions). *The optimized transformation of an expression e w.r.t. a set of variables V, denoted $tr(V, e)$, is defined as follows (the transformation share is as in Definition 7):*

$$
\begin{aligned}
tr(V, x) &= x \\
tr(V, s(\overline{e_n})) &= share(S, s'(tr(V \cup S, e_1), \ldots, tr(V \cup S, e_n))) \\
&\quad where \qquad S = multocc(\overline{e_n}) \setminus V \\
&\qquad\qquad s'(\overline{t_n}) = \begin{cases} \texttt{(}s\ t_1\ \ldots\ t_n\texttt{)} & ,\ if\ \ s \in \mathcal{F} \\ \texttt{(return (}s\ t_1\ \ldots\ t_n\texttt{))} & ,\ if\ \ s \in \mathcal{C} \end{cases} \\
tr(V, e_1 \,?\, e_2) &= \texttt{(mplus}\ tr(V, e_1)\ tr(V, e_2)\texttt{)} \\
tr(V, case\ e\ of\ \{\overline{p_n \to e_n}\}) &= share(S, \left\{ \begin{array}{ll} \texttt{(}tr(V \cup S, e)\ \texttt{>>=}\ \lambda \texttt{x} \to & \\ \texttt{case x of} & \\ \quad p_1 & \to tr(V \cup S, e_1) \\ \quad \ldots & \\ \quad p_n & \to tr(V \cup S, e_n) \\ \quad \texttt{\_} & \to \texttt{mzero)} \end{array} \right\}) \\
&\quad where \quad \texttt{x}\ fresh \\
&\qquad\qquad S = (\textstyle\bigcup_{i=1}^{n} multocc(e, e_i)) \setminus V
\end{aligned}
$$

According to the idea that the additional argument of the transformation represents the set of variables for which sharing was already introduced, the initial value of the argument should be the empty set as expressed in the next definition.

Definition 11 (Optimized Transformation of Functions). *The optimized transformation of a function defined by a rule $f(\overline{x_n}) \to e$ is similar to Definition 8 but uses the transformation from Definition 10.*

$f\ x_1\ \ldots\ x_n\ \texttt{=}\ tr(\emptyset, e)$

5 Conclusions and Related Work

In this paper we presented a scheme to translate functional logic programs into pure Haskell programs. The difficulty of such a translation is the fact that non-deterministic results can occur in any level of a computation, i.e., arbitrarily deep inside data structures. This problem is solved by transforming all computations into monadic ones, i.e., all argument and result values of functions and data constructors have monadic types w.r.t. a "non-determinism monad", i.e. a `MonadPlus` instance. Furthermore, the monad must support explicit sharing in order to implement the sharing of potentially non-deterministic arguments, which is necessary for a non-strict functional logic language with call-time choice. As a result, we obtain target programs which are parametric w.r.t. the concrete evaluation monad, i.e., one can execute the same target code with different search strategies, choose between call-time choice or run-time choice for parameter passing, or add additional run-time information to implement specific tools.

Considering related work, many schemes to compile lazy functional logic programs into various target languages have been introduced. Due to the nature of these languages, former approaches can be categorized with respect to the target language: (a) schemes targeting a logic programming language (b) compiling to a

lazy functional language (c) generating code for a especially devised abstract machine (implemented in an imperative language, typically). Considering (a) there have been several attempts to target Prolog and make use of the logic features of that host language, e.g., the $\mathcal{TOY}$ system [16], and PAKCS [11]. With respect to the implementation presented here, a system based on Prolog can not easily support different search strategies simply because Prolog does not support them. On the other hand, Prolog implementations normally offer various constraint solvers, which can therefore be easily integrated in a functional logic system. Typically, however, these integrations suffer from the fact that constraint solvers for Prolog are implemented with respect to a strict semantics. The resulting issues with a lazy semantics make such an integration not as seamless as possible. With respect to (b) there have been various proposals to implement logic programming in a functional language. As discussed in detail in [8], most of these proposals do not adequately represent laziness. The exception to this is KiCS [7], which employs a different translation scheme to compile Curry to Haskell. In contrast to the scheme presented here, the current implementation of KiCS employs side effects for the implementation of logic features. Consequently, the resulting programs can not be optimized by standard Haskell compilers. In addition, the attempt to introduce a parallel search strategy to KiCS has failed due to the side effects. In contrast to our approach, however, KiCS provides sharing of deterministic expressions across non-deterministic computations [7]. Regarding (c), sharing across non-determinism is also provided by FLVM, the abstract machine described in [5], which is implemented in Java. The FLVM has undergone substantial changes from the implementation described in [5], and can still be considered to be in an experimental state. Finally, the MCC [18] is based on an abstract machine implemented in C. The MCC provides a programatic approach to support different search strategies, i.e., the Curry programmer can influence the search strategy by calling primitive operators provided in this system.

Bundles [15] improve laziness in purely functional non-deterministic computations similar to our translation of data types. The type for bundles is a transformed list data type restricted to the list monad without non-deterministic list elements. Nesting non-determinism inside constructors plays an essential role in achieving full abstraction in a semantics for constructor systems under run-time choice [17].

By representing non-determinism explicitly using monads, we can collect results of non-deterministic computations in a deterministic data structure which is called encapsulated search [6, 3]. The monadic formulation of lazy non-determinism provides a new perspective on the problems described in previous work on encapsulated search and possibilities for future work.

In a next step, we will implement the transformation scheme into a complete compiler for Curry in order to test it on a number of benchmarks. Although it is clear that one has to pay a price (in terms of execution efficiency) for the high-level parametric target code, initial benchmarks, presented in [8], demonstrate that the clean target code supports optimizations of the Haskell compiler so that the monadic functional code can compete with other more low level implementations. Based on such an implementation, it would be interesting to test it with various monad instances in order to try different search strategies, in particular, parallel strategies, or to implement support for run-time tools, like observation tools, de-

buggers etc. Furthermore, one could also use the monad laws of [8] together with our transformation scheme in order to obtain a verified implementation of Curry.

References

1. S. Antoy. Constructor-based conditional narrowing. In *Proc. of PPDP 2001*, pages 199–206. ACM Press, 2001.
2. S. Antoy. Evaluation strategies for functional logic programming. *Journal of Symbolic Computation*, 40(1):875–903, 2005.
3. S. Antoy and B. Braßel. Computing with subspaces. In M. Leuschel and A. Podelski, editors, *Proc. of PPDP 2007*, pages 121–30. ACM, 2007.
4. S. Antoy and M. Hanus. Overlapping rules and logic variables in functional logic programs. In *Proc. of the 22nd Int. Conference on Logic Programming (ICLP 2006)*, pages 87–101. Springer LNCS 4079, 2006.
5. S. Antoy, M. Hanus, J. Liu, and A. Tolmach. A virtual machine for functional logic computations. In *Proc. of the 16th Int. Workshop on Implementation and Application of Functional Languages (IFL 2004)*, pages 108–125. Springer LNCS 3474, 2005.
6. B. Braßel, M. Hanus, and F. Huch. Encapsulating non-determinism in functional logic computations. *Journal of Functional and Logic Programming*, 2004(6), 2004.
7. B. Braßel and F. Huch. The Kiel Curry system KiCS. In *Applications of Declarative Programming and Knowledge Management*, pages 195–205. Springer LNAI 5437, 2009.
8. S. Fischer, O. Kiselyov, and C. Shan. Purely functional lazy non-deterministic programming. In *Proc. of ICFP 2009*, pages 11–22. ACM, 2009.
9. J. González-Moreno, M. Hortalá-González, F. López-Fraguas, and M. Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming*, 40:47–87, 1999.
10. M. Hanus. Multi-paradigm declarative languages. In *Proc. of the Int. Conference on Logic Programming (ICLP 2007)*, pages 45–75. Springer LNCS 4670, 2007.
11. M. Hanus, S. Antoy, B. Braßel, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, and F. Steiner. PAKCS: The Portland Aachen Kiel Curry System. Available at `http://www.informatik.uni-kiel.de/~pakcs/`, 2008.
12. M. Hanus and C. Prehofer. Higher-order narrowing with definitional trees. *Journal of Functional Programming*, 9(1):33–75, 1999.
13. M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.8.2). Available at `http://www.curry-language.org`, 2006.
14. H. Hussmann. Nondeterministic algebraic specifications and nonconfluent term rewriting. *Journal of Logic Programming*, 12:237–255, 1992.
15. F. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. Bundles pack tighter than lists. In *Draft Proc. of Trends in Functional Programming 2007*, pages XXIV–1–XXIV–16, 2007.
16. F. López-Fraguas and J. Sánchez-Hernández. TOY: A multiparadigm declarative system. In *Proc. of RTA'99*, pages 244–247. Springer LNCS 1631, 1999.
17. F. J. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. A fully abstract semantics for constructor systems. In *RTA '09: Proc. of the 20th Int. Conference on Rewriting Techniques and Applications*, pages 320–334. Springer, 2009.
18. W. Lux and H. Kuchen. An efficient abstract machine for Curry. In K. Beiersdörfer, G. Engels, and W. Schäfer, editors, *Informatik '99 — Annual meeting of the German Computer Science Society (GI)*, pages 390–399. Springer, 1999.
19. S. Peyton Jones, editor. *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press, 2003.
20. P. Wadler. How to replace failure by a list of successes. In *Functional Programming and Computer Architecture*, pages 113–128. Springer LNCS 201, 1985.