From Functional Logic Programs to Purely Functional Programs Preserving Laziness*

Bernd Braßel and Sebastian Fischer

Christian-Albrechts-University of Kiel
{bbr,sebf}@informatik.uni-kiel.de

Abstract. Functional logic languages extend the setting of functional programming by non-deterministic choices, free variables and narrowing. Most existing approaches to simulate logic features in functional languages do not preserve laziness, i.e., they can only model strict logic programming like in Prolog. Lazy functional logic programming however, has interesting properties supporting a more declarative style of programming search without sacrificing efficiency.
We will present a recently developed technique to reduce all logic extensions to the single problem of generating unique identifiers. The impact of this reduction is a general scheme for compiling functional logic programs to lazy functional programs without side effects.
One of the design goals is that the purely functional parts of a program should not suffer from significant run-time overhead. Preliminary experiments confirm our hope for significant improvements of run-time performance even for non-deterministic programs but suggest further work to improve the memory requirements of those.

1 Introduction

The two main paradigms of declarative programming are functional and logic programming. Logic languages support non-deterministic choice, computing with partial information and search for solutions. Conceptually, the simplest way to provide logic features in functional languages is to express non-determinism using lists [28] but in principal any instance of the class `MonadPlus` can be used for this purpose, for examples see [19,20,24]. All these approaches model non-deterministic computations like in Prolog in the sense that all computations involving non-deterministic choices are *strict*. However, the *functional logic* paradigm is mainly motivated by the insight that laziness and non-deterministic search can be combined profitably. Especially, this combination allows to program in the expressive and intuitive *generate-and-test* style while effectively computing in the more efficient style of *test-of-generate* [16]. Recent applications of

* This work has been partially supported by the German Research Council (DFG) under grant Ha 2457/5-1.

this technique show that it is, for example, well suited for the demand-driven generation of test data [27,12]. Functional logic design patterns [3] illustrate further benefits of combining lazy functional and logic programming.

In order to realize the lazy functional logic paradigm several programming languages have been developed [14]. Curry [13,18] extends lazy functional programming by non-deterministic choice and free variables; the syntax closely corresponds to that of Haskell 98 [25]. Further lazy functional logic programming languages are Escher [21] and Toy [22]. Compiling lazy functional logic programs to various target languages, including C [23], C++ [21], Java [17,5], Prolog [2] and Haskell [7] has a long tradition.

This paper is mainly concerned with translating Curry to Haskell. Note, however, that the technique could easily be adapted to any of the functional logic languages mentioned above. Likewise, other lazy functional languages like Clean [26] would suit equally well as target language.

As shown previously [7,8], choosing a *lazy functional* target language has several advantages. Firstly, deterministic functions can be translated without imposing much overhead. Additionally, Haskell allows to implement sharing of computed values even across non-deterministic branches where existing implementations reevaluate shared expressions. Finally, in contrast to a logic target language, the explicit encoding of non-determinism allows more fine grained control of logic search.

The challenge of targeting Haskell, however, is to preserve the laziness of the source language which allows the efficient execution of programs written in the generate-and-test style. Therefore, previous approaches to non-determinism in Haskell [28,19,20,24] do not suffice.

The translation scheme developed in this paper features all advantages mentioned above. Additionally, it comprises the following advantages.

- It is the first scheme translating lazy functional logic programs to *purely* functional programs. Consequently, the resulting code can be *fully optimized*, in contrast to our previous approach [7] which relied on unsafe side effects for generating labels.
- The transformation is *simple* — one could even say "off-the-shelf" as the technique of uniquely identifying certain expressions is employed fairly often.

In Section 2.1 we describe the general idea of the transformation scheme in comparison to naive implementations of non-determinism using lists. The general transformation scheme is defined in Section 3, mainly Subsection 3.3. We relate the presented approach to other compilation schemes in Section 4, provide experimental comparisons in Section 5, and conclude in Section 6.

2 Informal Presentation of the Transformation

In this section we describe the problems of translating lazy functional logic programs and informally present the idea behind our solution. We first motivate *call-time choice* (Section 2.1) which describes the meaning of the interaction between laziness and non-determinism by means of two examples — one very simple, the other more elaborated. We then show that a naive encoding of non-determinism violates call-time choice (Section 2.2), present our approach to correctly implement it (Section 2.3), and finally draw special attention to finite failure (Section 2.4).

2.1 Non-Determinism and Laziness

The interaction of laziness and logic programming features — especially non-determinism — is non-trivial both semantically, as well as operationally, i.e., from the point of view of an implementation. Current lazy functional logic programming languages have agreed on a model coined *Call-Time Choice* that supports the intuition that variables are placeholders for *values* rather than possibly non-deterministic computations. An important consequence of this computational model is that a lazy (call-by-need) computation has the same results as an eager (call-by-value) computation of the same program (if the latter terminates).

The semantic consequences of call-time choice are usually illustrated with a variation of the following tiny program:

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

not, selfEq :: Bool → Bool

not True  = False
not False = True

selfEq b = iff b b

iff :: Bool → Bool → Bool
iff True  b = b
iff False b = not b
```

In functional logic programs, all matching rules of an operation[1] are applied non-deterministically. Whenever there are multiple matching rules we say that these rules *overlap*. This behaviour differs from functional languages where only the topmost matching rule of a function is applied. Hence, a call to `coin` has two non-deterministic results: `True` and `False`. The function `selfEq` checks whether

[1] When non-determinism is involved, we use the term *operation* rather than *function*.

its argument is equivalent to itself using the Boolean equivalence test `iff`. There are two call-by-value derivations for the goal `(selfEq coin)`:

```
selfEq coin
  | → selfEq True  → iff True  True  → True
  | → selfEq False → iff False False → not False → True
```

If we evaluate the same goal with call-by-name, we get two more derivations, both with a result that cannot be obtained with call-by-value:

```
selfEq coin → iff coin coin
  | → iff True coin → coin
        | → True
        | → False
  | → iff False coin → not coin
        | → not True  → False
        | → not False → True
```

In a call-by-need derivation of the goal, i.e., in a lazy programming language, `coin` is evaluated only once and the result of `(selfEq coin)` is `True`. Shared non-deterministic sub computations evaluate to the same value.

Current lazy functional logic programming languages conform to call-time choice semantics following the *principle of least astonishment* because — like well known from functional programming — lazy computations have the same results as eager ones. Therefore, call-time choice relates to call-by-need—not to call-by-name which would give too many results. This choice simplifies formal reasoning, e.g., the result of a call to `selfEq` will always be `True`. It seems the most reasonable choice in order to not confuse programmers. Imagine, e.g., a function `sort :: [Int] → [Int]` and the following definition that generates tests for this function:

```
sortTests :: ([Int],[Int])
sortTests = (l, sort l) where l free
```

Programmers will expect the second component of the pair to be a sorted reordering of the first. And they will still expect this behaviour if they use a non-deterministic *generator* for non-negative numbers instead of a free variable[2]:

```
sortTests :: ([Int],[Int])
sortTests = (l, sort l) where l = natList

natList :: [Int]
natList = []
natList = nat : natList

nat :: Int
nat = 0
nat = nat + 1
```

[2] It has been shown in [4] that logic variables can be simulated using such non-deterministic generators and this result relies on call-time choice semantics.

The `sort` function can be defined using the *test-of-generate* pattern [16]:

```
sort :: [Int] → [Int]
sort l | sorted p = p where p = permute l

sorted :: [Int] → Bool
sorted []       = True
sorted [_]      = True
sorted (m:n:ns) = m ≤ n && sorted (n:ns)

permute :: [a] → [a]
permute []     = []
permute (x:xs) = insert x (permute xs)

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

The definition of `sort` is only reasonable with call-time choice, i.e., if both occurrences of `p` (in the guard and in the right-hand side of `sort`) denote the same value. Otherwise, an arbitrary permutation of the input would be returned if some other permutation is sorted. Thanks to lazy evaluation, permutations need to be computed only as much as is necessary in order to decide whether they are sorted. For example, if the first two elements of a permutation are already out-of-order, then a presumably large number of possible completions can be discarded. Permutations of a list are computed recursively by inserting the head of a list at an arbitrary position in the permutation of the tail. The definition of `insert` uses overlapping rules to insert `x` either as new head or somewhere in the tail of a non-empty list.

Permutation sort demonstrates nicely the semantic effect of call-time choice and the operational effect of laziness which prunes away large parts of the search space by not evaluating unsorted permutations completely. Thus, it is a characteristic example for a search problem expressed in the more intuitive *generate-and-test* style but solved in the more efficient *test-of-generate* style. This pattern generalizes to other problems and is not restricted to sorting which is usually not expressed as a search problem.

2.2 Naive Functional Encoding of Non-Determinism

In a first attempt, we might consider to represent non-deterministic values using lists [28] and lift all operations to the list type. The program that computes `(selfEq coin)` would then be translated as follows:

```
goal :: [Bool]
goal = selfEq coin

coin :: [Bool]
```

```
coin = [True,False]

not, selfEq :: [Bool] → [Bool]
not bs = [ False | True ← bs ] ++ [ True | False ← bs ]

selfEq bs = iff bs bs

iff :: [Bool] → [Bool] → [Bool]
iff xs ys = [ y | True  ← xs, y ← ys ]
         ++ [ y | False ← xs, y ← not ys ]
```

However, this translation does not adhere to call-time choice semantics because argument variables of functions denote possibly non-deterministic computations rather than values. For example, the argument `bs` of `selfEq` represents all non-deterministic results of this argument and the function `iff` might choose different values for each of its arguments. Consequently, the result of evaluating `goal` is `[True,False,False,True]` which resembles a call-by-name derivation of the corresponding functional logic program rather than call-by-need.

In order to model call-time choice, we could translate all functions such that they take deterministic arguments and use the list monad to handle non-determinism. The same example would then be translated as follows (the definition of `coin` is unchanged):

```
goal :: [Bool]
goal = do { b ← coin; selfEq b }

not, selfEq :: Bool → [Bool]
not True  = return False
not False = return True

selfEq b = iff b b

iff :: Bool → Bool → [Bool]
iff True  b = return b
iff False b = not b
```

Here, the value of `goal` is `[True,True]` as in a call-by-value derivation of a functional logic program, i.e., it corresponds to call-time choice. Unfortunately, the resulting program is strict, e.g., the call to `coin` is evaluated before passing its result to the function `selfEq`. Strictness can lead to unexpected non-termination and performance problems due to unnecessary evaluations. In lazy functional logic programming, unnecessary evaluation often means *unnecessary search*. The consequence often is exponential overhead which is clearly unacceptable. We will see in Section 5 that for programs in generate-and-test style such overhead can be significantly reduced by laziness.

With a naive approach, and also with sophisticated optimizations [19,20,24], we have the choice between laziness and call-time choice, we cannot obtain both.

2.3 Combining Laziness and Call-Time Choice

In our approach to translating lazy functional logic programs we do not use lists to represent non-determinism. Instead, we introduce a new constructor `Choice :: ID → a → a → a` and use it to build trees of non-deterministic values. Of course, a constructor of this type cannot be defined in Haskell, but in order to keep the description of our transformation as simple as possible, we do not consider types in this paper. In an implementation we can introduce different choice constructors for every data type.

The type `ID` in the first argument of `Choice` is an abstract type with the following signature:

```
type ID
instance Eq ID
initID :: ID
leftID, rightID :: ID → ID
```

The functions `leftID` and `rightID` compute unique identifiers from a given identifier and are used to pass unique identifiers to every part of the computation that needs them. In order to ensure that the generated identifiers are indeed unique, the shown functions need to satisfy specific properties:

- `leftID` and `rightID` must not yield the same identifier for any arguments,
- they never yield an identifier equal to `initID`, and
- both functions yield different results when given different arguments.

More formally, we can state that `leftID` and `rightID` have disjoint images that do not contain `initID`, and are both injective. In the syntax of QuickCheck [10], these properties read as follows:

```
λi j → leftID i /= rightID j

λi → initID /= leftID i && initID /= rightID i

λi j → i /= j ⟹ leftID i /= leftID j && rightID i /= rightID j
```

A possible implementation of `ID` uses positive integers of unbounded size:

```
type ID = Integer  -- positive

initID :: ID
initID = 1

leftID, rightID :: ID → ID
leftID  i = 2*i
rightID i = 2*i + 1
```

This implementation satisfies the given properties for all positive integers. In fact, the choice of `1` in the definition of `initID` is arbitrary—any positive integer would suffice. This implementation is not perfect because the generated identifiers grow rapidly and many integers might not be used as identifiers depending on how the functions `leftID` and `rightID` are used. However, it is purely functional and serves well as a prototype implementation. There are more efficient implementations [6] that make selected use of side effects without sacrificing compiler optimizations.

Unique identifiers are crucial in our approach to translate lazy functional logic programs because they allow to detect sharing of non-deterministic choices. If the result of a computation contains occurrences of `Choice` with the same identifier, the same alternative of both choices needs to be taken when computing the (functional logic) *values*[3] of this expression. In order to label non-deterministic choices with unique identifiers, we need to pass them to every position in the program that eventually performs a non-deterministic choice. As a first example, we consider the translation of `(selfEq coin)` in our approach:

```
goal :: ID → Bool
goal i = selfEq (coin i)

coin :: ID → Bool
coin i = Choice i True False

not, selfEq :: Bool → Bool
not True           = False
not False          = True
not (Choice i x y) = Choice i (not x) (not y)

selfEq b = iff b b

iff :: Bool → Bool → Bool
iff True           z = z
iff False          z = not z
iff (Choice i x y) z = Choice i (iff x z) (iff y z)
```

We pass an identifier to the operations `goal` and `coin` because they either directly create a `Choice` or call an operation which does. The functions `selfEq`, `iff`, and `not` do not need an additional parameter. We only have to extend their pattern matching to handle choices. If a value constructed by `Choice` is demanded, we return a choice with the same identifier and reapply the function to the different alternatives to compute the alternatives of the result. With these definitions `(goal initID)` evaluates to the following result (assuming `initID` yields `1`).

```
Choice 1 (Choice 1 True False) (Choice 1 False True)
```

This result can be interpreted as `Choice 1 True True` because for all occurrences of `False` we would need to take once a left branch and once a right branch of a

[3] We define the computation of functional logic values in Section 3.4.

`Choice` labeled with `1`. In our approach, however, choices with the same label are constrained to take the same branch when computing non-deterministic results. The invalid branches of the inner choices are, hence, pruned away. As a result, we obtain call-time choice semantics without sacrificing laziness: `coin` is evaluated by `iff` — not before passing it to `selfEq`. Moreover, the computations leading to the invalid results `False` are never evaluated (see Section 3.4).

A more complex example is the translation of `permute` (see Section 2.1):

```
permute :: ID → [a] → [a]
permute _ []     = []
permute i (x:xs) = insert (leftID i) x (permute (rightID i) xs)
permute i (Choice il xs ys) = Choice il (permute i xs) (permute i ys)

insert :: ID → a → [a] → [a]
insert i x []     = [x]
insert i x (y:ys) =
  Choice (leftID i) (x:y:ys) (y : insert (rightID i) x ys)
insert i x (Choice il xs ys) =
  Choice il (insert i x xs) (insert i x ys)
```

Both functions take an identifier as additional argument because they either directly create a `Choice` or call an operation which does and both functions make use of `leftID` and `rightID` to generate new identifiers that are passed down to sub computations.

2.4 Failing Computations

Non-determinism is only one means to model search in (functional-)logic programs. The other one is (finite) failure. Unlike in functional programs, where failing computations are considered programming errors, a functional logic programmer uses incomplete patterns or guards to restrict search. For example, the `sort` function introduced in Section 2.1 uses a guard that fails for unsorted lists in order to constrain the set of results to sorted permutations of the input.

In order to model failing computations, we introduce an additional special constructor `Failure` — again ignoring types in the scope of this description. Similar to the special rules for `Choice`, every pattern matching needs to be extended with a rule for `Failure`. For example, consider the transformed version of `sort`[4]:

```
sort :: ID → [Int] → [Int]
sort i l = guard (sorted p) p where p = permute i l

guard :: Bool → a → a
guard True           z = z
guard (Choice i x y) z = Choice i (guard x z) (guard y z)
guard _              _ = Failure
```

[4] We omit the definitions of `(≤)` and `(&&)`.

```
sorted :: [Int] → Bool
sorted []                 = True
sorted [_]                = True
sorted (m:n:ns)           = m ≤ n && sorted (n:ns)
sorted (Choice i xs ys)   = Choice i (sorted xs) (sorted ys)
sorted (m:Choice i xs ys) = Choice i (sorted (m:xs)) (sorted (m:ys))
sorted _                  = Failure
```

We introduce a function `guard` to express guards in functional logic programs. This function returns its second argument if the first argument is `True`. Additionally, we have to add a rule to handle `Choice` in the first argument. We add a default case for all patterns that have not been matched to every function to return `Failure` in case of a pattern match error and to propagate such errors. Note that in the definition of the predicate `sorted` we need to handle `Choice` at every position in the nested pattern for lists with at least two elements.

3 Formal Definition of Transformation

In this section we define our program transformation formally. We introduce necessary notation (Sections 3.1 and 3.2), formalize the transformation (Section 3.3), and define the computation of non-deterministic *values* (Section 3.4).

3.1 Preliminaries

We consider functional logic programs to be constructor-based term rewriting systems. For this we consider a *constructor-based signature* Σ as a disjoint union of two sets of symbols $C \cup F$ along with a mapping from each symbol to a natural number, called the symbol's *arity*. We will write $s^n \in \Sigma$ to denote that Σ contains the symbol s and that the arity of s is n. Elements of the sets C and F are called *constructor* and *function symbols*, respectively. We will use the symbols $c, c_1, \ldots, c_n$ for constructor symbols, $f, g, h, f_1, \ldots, f_n$ for function symbols and $s, s_1, \ldots, s_n$ for arbitrary symbols in $C \cup F$.

In general we use the notation $\overline{o_n}$ to denote a *sequence of objects* $o_1, \ldots, o_n$. If the exact length and elements of the sequence are arbitrary we may write $\overline{o}$.

Terms are constructed from constructor-based signatures in the usual inductive way. There are several important sets of terms. Elements of the set of *variables* X are denoted by $x, y, z, x_1, \ldots, x_n$ and the set of *values* V is defined by $V \ni v ::= x \mid c(\overline{v_n})$ where $x \in X$ and $c^n \in C$. We consider X to be fixed in the following. Finally the set of *expressions* E is defined by $E \ni e ::= x \mid s(\overline{e_n})$ where $x \in X$ and $s^n \in \Sigma$.

A special class of terms is called *linear*. In a linear term every variable appears at most once. The *sub terms* of a term t are denoted by $sub(t)$ and can be

defined inductively by $sub(x) = \{x\}$; $sub(s(\overline{e_n})) = \{s(\overline{e_n})\} \cup \bigcup_{1 \leq i \leq n} sub(e_i)$. Likewise, the set of *variables* occurring in a term t, denoted by $var(t)$, is defined by $var(x) = \{x\}$; $var(s(\overline{e_n})) = \bigcup_{1 \leq i \leq n} var(e_i)$.

3.2 Programs and Operations

A *program* over a given signature Σ is a sequence of *rules*. Each rule is of the form $f(\overline{v}) \to e$ where $f(\overline{v})$ is linear. The set of all programs over a signature Σ is denoted by P_Σ but we can often simply write P when Σ is clear from the context or arbitrary. Elements of P will be denoted as p, p'. For a program p and a rule $r \in p$ where $r = f(\overline{v}) \to e$ we say that r is *a rule defining* (the operation) f (in p). The sequence of all rules defining f in p is denoted by $p(f)$.

An important subclass of programs are *uniform programs*. In a uniform program p for all rules $l \to r \in p$ holds $var(r) \subseteq var(l)$. Furthermore, the only non-deterministic operation is the binary $?^2$ defined by:

```
x ? y → x
x ? y → y
```

All other operations are defined either by a single rule without pattern matching or by a flat pattern matching on the last argument without any overlap. Formally, this means for any $f^n \neq ?^2$:

$$\begin{aligned} & p(f) = f(\overline{x_n}) \to e \\ \vee\ & p(f) = f(\overline{x_{n-1}}, c_1(\overline{y})) \to e_1, \ldots, f(\overline{x_{n-1}}, c_i(\overline{z})) \to e_i \end{aligned}$$

where the constructors $\overline{c_i}$ are pairwise different.

It is well known that any functional logic program can be transformed into an equivalent uniform program [1]. For this it is necessary to redefine overlapping rules using $?^2$ and eventually adding auxiliary functions if the overlap is not trivial. For example the operation `insert` from Section 2.1 is redefined as:

```
insert  x xs     = (x:xs) ? insert2 x xs
insert2 x (y:ys) = y : insert x ys
```

Complex pattern matching also requires the introduction of fresh operations. For example `sorted` from Section 2.1 is redefined as:

```
sorted []     = True
sorted (m:xs) = sorted2 m xs

sorted2 _ []     = True
sorted2 m (n:ns) = m ≤ n && sorted (n:ns)
```

Some arguments are swapped like those of `iff` (Section 2.1). Free variables are simulated by generators as discussed in Section 2.1 and justified in [4].

The simple structure of uniform programs allows us to concisely define our transformation in the following subsection.

3.3 The Transformation of Programs

For the following, assume a given uniform program p over a signature Σ. We assume that Σ does not contain any of those symbols which we want to add as discussed in Section 2.3. We denote the set of these symbols by

$$S := \{\texttt{hnf}^1, \texttt{leftID}^1, \texttt{rightID}^1, \texttt{initID}^0, \texttt{Choice}^3, \texttt{Failure}^0\}$$

In this section we define how to produce a (purely functional) program p' over a signature Σ'.

One of the design goals of the transformation is that purely functional computations should be as efficient as possible. To achieve this we have to distinguish between purely functional and (potentially) non-deterministic operations. A necessary requirement for an operation to be non-deterministic is that it depends on the operation $\texttt{?}^2$. In other words, it either calls $\texttt{?}^2$ directly or calls a function depending on $\texttt{?}^2$. Formally, the set of non-deterministic operations $N \subseteq \Sigma$ is the smallest set such that

$$N := \{\texttt{?}^2\} \cup \{f \mid \exists l \to r \in p(f) : \exists g(\overline{e_n}) \in \mathit{sub}(r) : g^n \in N\}$$

All elements of N are extended with an extra argument to form the new signature $\Sigma' := S \cup \Sigma \setminus N \cup \{f^{n+1} \mid f^n \in N\}$.

One of the main concepts discussed in Section 2.3 is that each non-deterministic sub expression is extended by a *unique identifier* generated by `leftID`, `rightID` and `initID`. For this let i be an expression of type `ID`, i.e., an expression yielding an identifier at run time. Then the function $\mathit{fresh}_n(i)$ generates an expression yielding a different identifier from i for each natural number n. This is achieved by adding n times the function `rightID` and finally `leftID`.[5]

$$\mathit{fresh}_n(i) = \texttt{leftID}(\texttt{rightID}^n(i))$$

The next definition covers the transformation of expressions. It adds a given expression i of type `ID` for each call to an operation in N.

$$\begin{aligned} tr(i, x) &= x \\ tr(i, s(\overline{e_n})) &= \begin{cases} s(\overline{tr(i_n, e_n)}) & \text{, if } s^n \notin N \\ s(i_{n+1}, \overline{tr(i_n, e_n)}) & \text{, if } s^n \in N \end{cases} \\ &\text{where} \\ \overline{i_{n+1}} &= \overline{\mathit{fresh}_{n+1}(i)} \end{aligned}$$

We are now ready to transform the rules defining an operation $f \neq \texttt{?}^2$. Each rule is transformed by an application of $tr(\texttt{i}, \cdot)$ to both the left and the right

[5] Note that we are quite wasteful in the generation of identifiers. We do so for simplicity; a transformation generating a minimal amount of calls to `leftID` and `rightID` is straightforward by counting non-deterministic sub terms.

hand side of the rule where $\mathtt{i}$ is a fresh variable not occurring anywhere in p. For operations with matching rules the additional rules to lift the `Choice` constructor (see Section 2.3) and to produce `Failure` (Section 2.4) are added.

$$p'(f) := \left\{ \begin{array}{ll} tr(\mathtt{i}, l) \to tr(\mathtt{i}, r) & , \; p(f) = l \to r, l = f(\overline{x_n}) \\ \left. \begin{array}{l} \overline{tr(\mathtt{i}, l_n) \to tr(\mathtt{i}, r_n)}, \\ f(\overline{x_m}, \mathtt{Choice}(x, y, z)) \\ \quad \to \mathtt{Choice}(x, f(\overline{x_m}, y), f(\overline{x_m}, z)), \\ f(\overline{x_m}, x) \to \mathtt{Failure} \end{array} \right\} & , \; \begin{array}{l} p(f) = \overline{l_n \to r_n}, \\ l_1 = f(\overline{x_m}, c(\overline{y})) \end{array} \end{array} \right.$$

The operation $?^2$ is replaced by $?^3$ which introduces the constructor `Choice`.

$$?(i, x, y) = \mathtt{Choice}(i, x, y)$$

Finally, the transformed program is extended by the definitions of the external operations `initID`, `leftID`, `rightID` and `hnf`. Possible implementations of `initID`, `leftID` and `rightID` were discussed in Section 2.3. The definition and application of `hnf` is the topic of the following subsection.

3.4 Evaluation to Head Normal Form and Transformation of Goals

In Section 2.3 we have seen that the transformed program yields terms of the form `Choice 1 (Choice 1 True False) (Choice 1 False True)` where the only valid solution is `True` (computed in two different ways). We will now define a function to abstract the admissible values from a given tree of choices.

```
hnf :: a → [a]
hnf ct = hnf' [] ct
  where hnf' choices (Choice i x y) = case lookup i choices of
              Just b  → hnf' choices (if b then x else y)
              Nothing → hnf' ((i,True) :choices) x
                      ++ hnf' ((i,False):choices) y
        hnf' _ Failure = []
        hnf' _ v       = [v]
```

The reader may check that indeed `hnf` applied on the tree of choices given above evaluates to `[True,True]`.

It is unfortunate, however, that this version of `hnf` fixes the search strategy. In this case, `hnf` models depth first search. A nice opportunity provided by our approach is to let the user define the search strategy by different traversals of a representation of the search space. For this we first define the type representing *search trees*.

```
data SearchTree a = Value a
                  | Branch (SearchTree a) (SearchTree a)
                  | Stub
```

Now we can define an `hnf` function with the type `a` $\rightarrow$ `SearchTree a`.

```
hnf :: a → SearchTree a
hnf ct = hnf' [] ct
  where hnf' choices (Choice i x y) = case lookup i choices of
              Just b  → hnf' choices (if b then x else y)
              Nothing → Branch (hnf' ((i,True) :choices) x)
                               (hnf' ((i,False):choices) y)
        hnf' _ Failure = Stub
        hnf' _ v       = Value v
```

This function basically replaces `Choice` constructors with `Branch` constructors and prunes away invalid branches when matching a `Choice` constructor with a label that has already been processed.

Different search strategies can now be defined as tree traversals. We only give the definition of depth first search. See [8] for a definition of breadth first search.

```
df :: SearchTree a → [a]
df (Value v)      = [v]
df (Branch t1 t2) = df t1 ++ df t2
df Stub           = []
```

Having defined `hnf`, the final step of our transformation is to translate the goals given by the user, e.g., as the body of a function `goal` or on an interactive command line environment. For any search strategy *st* a given goal e is then translated to `(`*st* `(hnf (`tr`(initID,`e`)))`.[6]

4 Related Work

The extension of lazy functional programming by logic features has by now a long tradition both in theory and application. A recent survey of the state of the art is given in [14].

There are several compilers for Curry which all follow one of three basic approaches. Compiling to *logic* languages like the PAKCS system [15], to *functional* languages like our former approach [8], or devising a *new abstract machine from scratch* [23,5]. Orthogonal to the development of a compiler for a complete language, several projects endeavor to extend functional programming by implementing *libraries* to support logic programming [19,20,24]. Each of these approaches has its own advantages and drawbacks.

Naturally, compiling to logic programming languages benefits from the efficient implementation of non-deterministic search. In addition, the target language

[6] Curry implementations yield complete normal forms instead of head normal forms. Normal-form computation is orthogonal to our new approach to labeling and discussed elsewhere [8].

will often feature further opportunities, e.g., to integrate constraint solvers [11]. Three main disadvantages, however, lead to handicaps with respect to both the performance of the average program and the class of applications that can be realized.

1. The *lazy evaluation of functions* has a substantial portion of *interpretation* in a logic programming language. Since in practice most functional logic programs feature more functional than logic code[7] and because laziness is very important for efficient search this results in cut backs for average performance.
2. As languages for logic programming are conceptually strict, there is no sensible notion of *sharing across non-determinism*. Sharing purely functional evaluations for each branch of a search space is, however, an important advantage for performance [7]. Especially, tabling techniques are not easily adapted to lazy semantics.
3. Since the resulting programs reuse the search facilities of Prolog it is impossible to support *control of the search*. However, the recent examples of implementing test tools [9] show that the opportunity to define sophisticated search strategies can be vital for the successful development of applications.

Obviously, when devising an abstract machine from scratch and compiling to a low level language like C or Java the implementation can support a detailed control of the search. Currently, there exists one mature implementation compiling to C, the MCC [23], and an experimental one to Java [5]. The main drawback of such an enterprise is that most of the knowledge amassed about declarative languages has to be reimplemented. Although an admirable effort in this regard has been realized for the MCC, we think that in the long run it is more promising to benefit from the development of mature Haskell compilers like GHC. This includes the integration of concurrent programming, e.g. by STMs, and experiment with parallel search strategies. In contrast to the MCC the approach compiling to Java [5] also includes — like ours — sharing across non-determinism. There, however, no effort has been taken to implement any of the well known techniques to optimize declarative programs.

Libraries for logic programming in Haskell all share the same drawback in comparison to our approach: they do *not preserve laziness* for logic operations. Therefore, lazy functional logic programs cannot directly be translated to Haskell employing such a library. In contrast, it is one of the main insights of the research in functional logic programming languages that laziness and non-deterministic search can be combined profitably [16]. The next subsection shows by example that integrating laziness can significantly improve the performance of search.

[7] This is more than just an observation about the current style of functional logic programming. It is in the essence of non-determinism that it is more expressive while being more computationally demanding in general. Therefore, sensible optimization techniques would often transform logic code to functional code when possible. Consequently, a good implementation of pure functions is crucial.

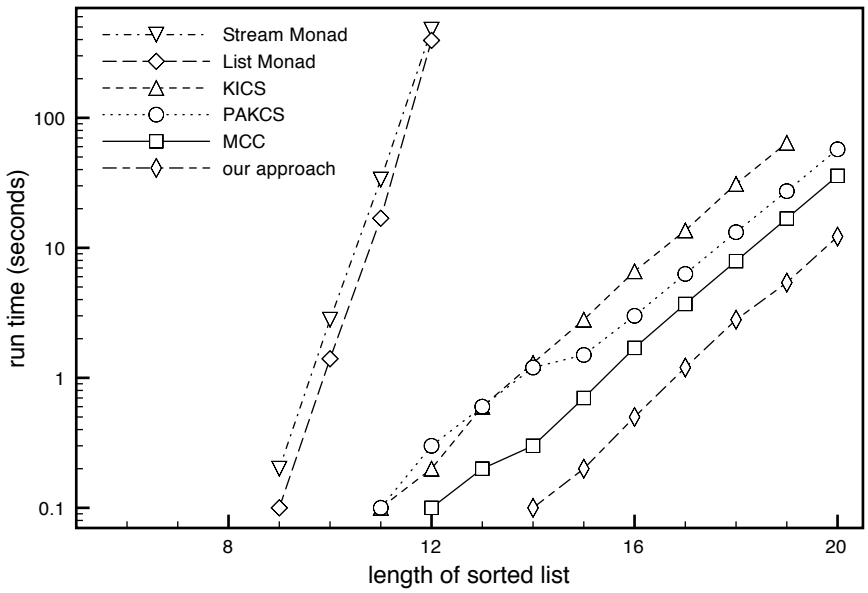

Fig. 1. Run time of Permutation Sort

5 Experimental Comparison

We have tested our approach to transform lazy functional logic programs in order to show a) the benefit of preserving laziness compared to monadic approaches in Haskell, and of b) using optimizations of an up-to-date Haskell compiler compared to other implementations of Curry.

We have applied the `sort` function defined in Section 2.1 to lists of different length and measured the run time for each investigated system. We have investigated two monadic approaches to model non-determinism: the list monad and a fair monad of streams [20] which we will call "stream monad" in the following. Additionally, we have compared our approach to existing Curry systems — especially, to the Kiel Curry System KiCS, our former approach, that uses a translation scheme similar to the one presented in this paper but uses side effects that prohibit optimizations on the generated Haskell code. We have used optimizations whenever available (optimizations have not been available in PAKCS and KiCS). The reported CPU times are user time in seconds, measured by the `time` command on a 2.2 GHz Intel Core 2 Duo Mac Book. The results of our experiments are depicted in Figure 1. The run times of the `sort` function are plotted at a logarithmic scale — they range from fractions of a second to several minutes. We clearly see that permutation sort is an exponential time algorithm even if laziness prevents the computation of unnecessary sub permutations.

However, the lazy implementations clearly outperform the strict monadic versions of `sort`. The list monad is even faster than the more sophisticated stream monad — probably because fairness is not an issue here and any sophistication to achieve it is worthless effort for this example. The missing laziness is a problem for the monadic implementations. They need minutes for lists of length 12 while the run time of the lazy implementations is insignificant for such lists.

We experienced the impure version of KiCS to be about two times slower than PAKCS, which is itself two times slower than MCC, which is three times slower than an optimized Haskell program that is transformed according to the transformation presented in this paper. A speedup of 12 compared to our old implementation of KiCS is quite encouraging, although the memory requirements of our approach should dampen our optimism. Applying permutation sort to a list of 20 elements resulted in a memory allocation failure in the old version of KiCS and also in our new approach the memory requirements increase with the length of the sorted list. The MCC runs permutation sort in constant space, which hints at a problem with garbage collection that is yet to be solved.

6 Conclusions and Future Work

We have presented a new scheme to translate lazy functional logic programs into purely functional programs. To the best of our knowledge, we are the first to provide such a translation that preserves laziness without using side effects. As a consequence, the resulting programs can be fully optimized and preliminary experiments show that the resulting code performs favorably compared to other compilation techniques. Moreover, our transformation scheme adds almost no overhead to purely functional parts of a program: only patterns have to be extended with rules for choices and failure; unique identifiers only need to be passed to operations that may cause non-determinism.

We do not prove the correctness of our approach in this paper. Conceptually, however, the approach is similar to a previously presented transformation scheme implemented in the Curry compiler KiCS, which employs side effects. The correctness of that scheme has been shown previously [8] also taking into account the side effects.

We also do not discuss how to translate higher-order functions due to lack of space. Higher-order functions can be integrated in a way similar to the approach presented in [12]. This integration does not involve defunctionalization but reuses the higher-order features of the target language.

For future work we have to *implement the compilation scheme* described in Section 2.3. As the scheme is an extension of the one used in our earlier system [7] (and even simpler) we should be able to reuse much of our former work. This time, however, we plan to write the compiler in Curry rather than in Haskell.

While the run times for our compilation scheme look promising there is still work to do with regard to *memory usage*. Up to now the MCC system stands alone when regarding the combination of time and space efficiency.

A further topic of improvement concerns *constraint programming*. Our approach so far is not up to par with that of the PAKCS or the MCC system. PAKCS provides an interface to constraint solvers of the target language Prolog, MCC implements its own constraint solvers. Constraint programming could be integrated in our approach by interfacing external solvers or implementing solvers in Haskell. The details of a general framework for constraint programming in purely functional languages are still investigated in ongoing research.

References

1. S. Antoy. Evaluation strategies for functional logic programming. *Journal of Symbolic Computation*, 40(1):875–903, 2005.
2. S. Antoy and M. Hanus. Compiling multi-paradigm declarative programs into Prolog. In *Proc. International Workshop on Frontiers of Combining Systems (FroCoS'2000)*, pages 171–185. Springer LNCS 1794, 2000.
3. S. Antoy and M. Hanus. Functional logic design patterns. In *Proc. of the 6th International Symposium on Functional and Logic Programming (FLOPS 2002)*, pages 67–87. Springer LNCS 2441, 2002.
4. S. Antoy and M. Hanus. Overlapping rules and logic variables in functional logic programs. In *Proceedings of the 22nd International 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 International Workshop on Implementation and Application of Functional Languages (IFL 2004)*, pages 108–125. Springer LNCS 3474, 2005.
6. Lennart Augustsson, Mikael Rittri, and Dan Synek. On generating unique names. *Journal of Functional Programming*, 4(1):117–123, 1994.
7. B. Braßel and F. Huch. The Kiel Curry system KiCS. In *Proc. 17th International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2007) and 21st Workshop on (Constraint) Logic Programming (WLP 2007)*, pages 215–223. Technical Report 434, University of Würzburg, 2007.
8. Bernd Braßel and Frank Huch. On a tighter integration of functional and logic programming. In Zhong Shao, editor, *APLAS*, volume 4807 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 2007.
9. J. Christiansen and S. Fischer. EasyCheck - test data for free. In *Proc. of the 9th International Symposium on Functional and Logic Programming (FLOPS 2008)*, pages 322–336. Springer LNCS 4989, 2008.
10. Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *ICFP '00: Proceedings of the ACM Internatioal Conference on Functional Programming*, pages 268–279. ACM, 2000.
11. A.J. Fernández, M.T. Hortalá-González, F. Sáenz-Pérez, and R. del Vado-Vírseda. Constraint functional logic programming over finite domains. *Theory and Practice of Logic Programming (to appear)*, 2007.

12. S. Fischer and H. Kuchen. Data-flow testing of declarative programs. In *Proc. of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP 2008)*, pages 201–212. ACM Press, 2008.
13. M. Hanus. A unified computation model for functional and logic programming. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages (Paris)*, pages 80–93, 1997.
14. M. Hanus. Multi-paradigm declarative languages. In *Proceedings of the International Conference on Logic Programming (ICLP 2007)*, pages 45–75. Springer LNCS 4670, 2007.
15. 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/`, 2007.
16. M. Hanus and P. Réty. Demand-driven search in functional logic programs. Research report rr-lifo-98-08, Univ. Orléans, 1998.
17. M. Hanus and R. Sadre. An abstract machine for curry and its concurrent implementation in java. *Journal of Functional and Logic Programming*, 1999(6), 1999.
18. M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.8.2). Available at `http://www.curry-language.org`, 2006.
19. Ralf Hinze. Deriving backtracking monad transformers. In Phil Wadler, editor, *Proceedings of the 2000 International Conference on Functional Programming, Montreal, Canada, September 18-20, 2000*, pages 186–197, sep 2000.
20. O. Kiselyov. Simple fair and terminating backtracking monad transformer. Available at: `http://okmij.org/ftp/Computation/monads.html#fair-bt-stream`, October 2005.
21. J.W. Lloyd. Declarative programming in escher. Technical report cstr-95-013, University of Bristol, 1995.
22. F. López-Fraguas and J. Sánchez-Hernández. TOY: A multiparadigm declarative system. In *Proc. of RTA'99*, pages 244–247. Springer LNCS 1631, 1999.
23. 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.
24. Matthew Naylor, Emil Axelsson, and Colin Runciman. A functional-logic library for wired. In *Haskell '07: Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, pages 37–48, New York, NY, USA, 2007. ACM.
25. S. Peyton Jones, editor. *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press, 2003.
26. R. Plasmeijer and M. van Eekelen. Clean language report version 2.0, 2002. `http://clean.cs.ru.nl/CleanExtra/report20/`.
27. Colin Runciman, Matthew Naylor, and Fredrik Lindblad. SmallCheck and Lazy SmallCheck: Automatic exhaustive testing for small values. In *Haskell '08: Proceedings of the first ACM SIGPLAN symposium on Haskell*, pages 37–48. ACM, 2008.
28. P. Wadler. How to replace failure by a list of successes. In *Functional Programming and Computer Architecture*, pages 113–128. Springer LNCS 201, 1985.