KiCS2: A New Compiler from Curry to Haskell

Bernd Braßel Michael Hanus Björn Peemöller Fabian Reck

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

Abstract. In this paper we present our first steps towards a new system to compile functional logic programs of the source language Curry into purely functional Haskell programs. Our implementation is based on the idea to represent non-deterministic results as values of the data types corresponding to the results. This enables the application of various search strategies to extract values from the search space. We show by several benchmarks that our implementation can compete with or outperform other existing implementations of Curry.

1 Introduction

Functional logic languages integrate the most important features of functional and logic languages (see [8,24] for recent surveys). In particular, they combine higher-order functions and demand-driven evaluation from functional programming with logic programming features like non-deterministic search and computing with partial information (logic variables). The combination of these features has led to new design patterns [6] and better abstractions for application programming, e.g., as shown for programming with databases [14,18], GUI programming [21], web programming [22,23,26], or string parsing [17].

The implementation of functional logic languages is challenging since a reasonable implementation has to support the operational features mentioned above. One possible approach is the design of new abstract machines appropriately supporting these operational features and implementing them in some (typically, imperative) language, like C [32] or Java [9,27]. Another approach is the reuse of already existing implementations of some of these features by translating functional logic programs into either logic or functional languages. For instance, if one compiles into Prolog, one can reuse the existing backtracking implementation for non-deterministic search as well as logic variables and unification for computing with partial information. However, one has to implement demand-driven evaluation and higher-order functions [5]. A disadvantage of this approach is the commitment to a fixed search strategy (backtracking).

If one compiles into a non-strict functional language like Haskell, one can reuse the implementation of lazy evaluation and higher-order functions, but one has to implement non-deterministic evaluations [13,15]. Although Haskell offers list comprehensions to model backtracking [36], this cannot be exploited due to the specific semantical requirements of the combination of non-strict and non-deterministic operations [20]. Thus, additional implementation efforts are necessary like implementation of shared non-deterministic computations [19].

Nevertheless, the translation of functional logic languages into other high-level languages is attractive: it limits the implementation efforts compared to an implementation from scratch and one can exploit the existing implementation technologies, provided that the efforts to implement the missing features are reasonable.

In this paper we describe an implementation that is based on the latter principle. We present a method to compile programs written in the functional logic language Curry [28] into Haskell programs based on the ideas shown in [12]. The difficulty of such an implementation is the fact that non-deterministic results can occur in any place of a computation. Thus, one cannot separate logic computations by the use of list comprehensions [36], as the outcome of any operation could be potentially non-deterministic, i.e., it might have more than one result value. We solve this problem by an explicit representation of non-deterministic values, i.e., we extend each data type by another constructor to represent the choice between several values. This idea is also the basis of the Curry implementation KiCS [15,16]. However, KiCS is based on unsafe features of Haskell that inhibit the use of optimizations provided by Haskell compilers like GHC.[1] In contrast, our implementation, called KiCS2, avoids such unsafe features. In addition, we also support more flexible search strategies and new features to encapsulate non-deterministic computations (which are not described in detail in this paper due to lack of space).

The general objective of our approach is the support of flexible strategies to explore the search space resulting from non-deterministic computations. In contrast to Prolog-based implementations that use backtracking and, therefore, are incomplete, we also want to support complete strategies like breadth-first search, iterative deepening or parallel search (in order to exploit multi-core architectures). We achieve this goal by an explicit representation of the search space as data that can be traversed by various operations. Moreover, purely deterministic computations are implemented as purely functional programs so that they are executed with almost the same efficiency as their purely functional counterparts.

In the next section, we sketch the source language Curry and introduce a normalized form of Curry programs that is the basis of our translation scheme. Section 3 presents the basic ideas of this translation scheme. Benchmarks of our initial implementation of this scheme are presented in Section 4. Further features of our system are sketched in Section 5 before we conclude in Section 6.

2 Curry Programs

The syntax of the functional logic language Curry [28] is close to Haskell [35]. In addition, Curry allows free (logic) variables in conditions and right-hand sides of defining rules. In contrast to functional programming and similarly to logic programming, operations can be defined by overlapping rules so that they might yield more than one result on the same input. Such operations are also called *non-deterministic*. For instance, Curry offers a *choice* operation that is predefined by the following rules:

```
x ? _ = x
_ ? y = y
```

[1] `http://www.haskell.org/ghc/`

Thus, we can define a non-deterministic operation `aBool` by

```
aBool = True ? False
```

so that the expression "`aBool`" has two values: `True` and `False`.

If non-deterministic operations are used as arguments in other operations, a semantical ambiguity might occur. Consider the operations

```
xor True  False = True
xor True  True  = False
xor False x     = x

xorSelf x = xor x x
```

and the expression "`xorSelf aBool`". If we interpret this program as a term rewriting system, we could have the reduction

```
xorSelf aBool  →  xor aBool aBool  →  xor True aBool
               →  xor True False   →  True
```

leading to the unintended result `True`. Note that this result cannot be obtained if we use a strict strategy where arguments are evaluated before the operation calls. In order to avoid dependencies on the evaluation strategies and exclude such unintended results, González-Moreno et al. [20] proposed the rewriting logic CRWL 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 [29] where values of the arguments of an operation are determined before the operation is evaluated. Note that this does not necessarily require an eager evaluation of arguments. Actually, [1,31] define lazy evaluation strategies for functional logic programs with call-time choice semantics where actual arguments passed to operations are shared. Hence, we can evaluate the expression above lazily, provided that all occurrences of `aBool` are shared so that all of them reduce either to `True` or to `False`. The requirements of the call-time choice semantics are the reason why it is not simply possible to use list comprehensions or non-determinism monads for a straightforward implementation of functional logic programs in Haskell [19].

Due to these considerations, an implementation of Curry has to support lazy evaluation where operations can have multiple results and unevaluated arguments must be shared. This is a complex task, especially if we try to implement it directly on the level of source programs. Therefore, we perform some simplifications on programs before the target code is generated.

First of all, we assume that our programs *do not contain logic variables*. This assumption can be made since it has been shown [7] that logic variables can be replaced by non-deterministic "generators", i.e., operations that evaluate to all possible values of the type of the logic variable. For instance, a Boolean logic variable can be replaced by the generator `aBool` defined above.

Furthermore, we discuss our translation scheme only for *first-order* programs for the sake of simplicity. However, our implementation also supports higher-order features (see Section 4) by exploiting the corresponding features of Haskell.

Finally, we assume that the pattern matching strategy is explicitly encoded in individual matching functions. In contrast to [1], where the pattern matching strategy is

$e ::=$	x	x is a variable
	$\mid\ c(e_1, \ldots, e_n)$	c is an n-ary constructor symbol
	$\mid\ f(e_1, \ldots, e_n)$	f is an n-ary function symbol
	$\mid\ e_1 \,?\, e_2$	choice
$D ::=$	$f(x_1, \ldots, x_n) = e$	n-ary function f with a single rule
	$\mid\ f(c(y_1, \ldots, y_m), x_2, \ldots, x_n) = e$	matching rule for n-ary function f
		c is an m-ary constructor symbol
$P ::=$	$D_1 \ldots D_k$	

Fig. 1. Uniform programs (e: expressions, D: definitions, P: programs)

encoded in case expressions, we assume that each case expression is transformed into a new operation in order to avoid complications arising from the translation of nested case expressions. Thus, we assume that all programs are *uniform* according to the definition in Fig. 1.[2] There, the variables in the left-hand sides of each rule are pairwise different, and the constructors in the left-hand sides of the matching rules of each function are pairwise different. Uniform programs have a simple form of pattern matching: either a function is defined by a single rule without pattern matching, or it is defined by rules with only one constructor in the left-hand side of each rule, and in the same argument for all rules.[3] For instance, the operation `xor` defined above can be transformed into the following uniform program:

```
xor True  x = xor′ x
xor False x = x
xor′ False = True
xor′ True  = False
```

In particular, there are no overlapping rules for functions (except for the choice operation "`?`" which is considered as predefined). Antoy [3] showed that each functional logic program, i.e., each constructor-based conditional term rewriting system, can be translated into an equivalent unconditional term rewriting system without overlapping rules but containing choices in the right-hand sides, also called LOIS (limited overlapping inductively sequential) system. Furthermore, Braßel [11] showed the semantical equivalence of narrowing computations in LOIS systems and rewriting computations in uniform programs. Due to these results, uniform programs are a reasonable intermediate language for our translation into Haskell which will be presented in the following.

[2] A variant of uniform programs has been considered in [33] to define lazy narrowing strategies for functional logic programs. Although the motivations are similar, our notion of uniform programs is more restrictive since we allow only a single non-variable argument in each left-hand side of a rule. Uniform programs have also been applied in [37] to define a denotational analysis of functional logic programs.

[3] For simplicity, we require in Fig. 1 that the matching argument is always the first one, but one can also choose any other argument.

3 Compilation to Haskell: The Basics

3.1 Representing Non-deterministic Computations

As mentioned above, our implementation is based on the explicit representation of non-deterministic results in a data structure. This can easily be achieved by adding a constructor to each data type to represent a choice between two values. For instance, one can redefine the data type for Boolean values as follows:

```
data Bool = False | True | Choice Bool Bool
```

Thus, we can implement the non-deterministic operation `aBool` defined in Section 2 as:

```
aBool = Choice True False
```

If operations can deliver non-deterministic values, we have to extend the rules for operations defined by pattern matching so that they do not fail on non-deterministic argument values. Instead, they move the non-deterministic choice one level above, i.e., a choice in some argument leads to a choice in any result of this operation (this is also called a "pull-tab" step in [2]). For instance, the rules of the uniform operation `xor` shown above are extended as follows:

```
xor True           x = xor' x
xor False          x = x
xor (Choice x1 x2) x = Choice (xor x1 x) (xor x2 x)

xor' False          = True
xor' True           = False
xor' (Choice x1 x2) = Choice (xor' x1) (xor' x2)
```

The operation `xorSelf` is not defined by a pattern matching rule and, thus, need not be changed. If we evaluate the expression "`xorSelf aBool`", we get the result

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

How can we interpret this result? In principle, the choices represent different possible values. Thus, if we want to show the different values of an expression (which is usually the task of a top-level "read-eval-print" loop), we enumerate all values contained in the choices. These are `False`, `True`, `True`, and `False` in the result above. Unfortunately, this does not conform to the call-time choice semantics discussed in Section 2 which excludes a value like `True`. The call-time choice semantics requires that the choice of a value made for the initial expression `aBool` should be consistent in the entire computation. For instance, if we select the value `False` for the expression `aBool`, this selection should be made at all other places where this expression might have been copied during the computation. However, our initial implementation duplicates the initially single `Choice` into finally three occurrences of `Choice`.

We can correct this unintended behavior of our implementation by identifying different `Choice` occurrences that are duplicates of some single `Choice`. This can be easily done by attaching a unique identifier, e.g., a number, to each choice:

```
type ID = Integer
data Bool = False | True | Choice ID Bool Bool
```

Furthermore, we modify the `Choice` pattern rules so that the identifiers will be kept, e.g.,

```
xor (Choice i x1 x2) x = Choice i (xor x1 x) (xor x2 x)
```

If we evaluate the expression "`xorSelf aBool`" and assign the number `1` to the choice of `aBool`, we obtain the result

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

When we show the values contained in this result, we have to make *consistent* selections in choices with same identifiers. Thus, if we select the left branch as the value of the outermost `Choice`, we also have to select the left branch in the selected argument `(Choice 1 False True)` so that only the value `False` is possible here. Similarly, if we select the right branch as the value of the outermost `Choice`, we also have to select the right branch in its selected argument `(Choice 1 True False)` which yields the sole value `False`.

Note that each `Choice` occurring for the first time in a computation has to get its own unique identifier. For instance, if we evaluate the expression "`xor aBool aBool`", the two occurrences of `aBool` assign different identifiers to their `Choice` constructor (e.g., `1` for the left and `2` for the right `aBool` argument) so that this evaluates to

```
Choice 1 (Choice 2 False True) (Choice 2 True False)
```

Here we can make different selections for the outer and inner `Choice` constructors so that this non-deterministic result represents four values.

To summarize, our implementation is based on the following principles:

1. Each non-deterministic choice is represented by a `Choice` constructor with a unique identifier.
2. When matching a `Choice` constructor, the choice is moved to the result of this operation with the same identifier, i.e., a non-deterministic argument yields non-deterministic results for each of the argument's values.
3. Each choice introduced in a computation is supplied with its own unique identifier.

The latter principle requires the creation of fresh identifiers during a computation—a non-trivial problem in functional languages. One possibility is the use of a global counter that is accessed by unsafe features whenever a new identifier is required. Unfortunately, unsafe features inhibit the use of optimization techniques developed for purely functional programs and make the application of advanced evaluation and search strategies (e.g., parallel strategies) more complex. Therefore, we avoid unsafe features in our implementation. Instead, we thread some global information through our program in order to supply fresh references at any point of a computation. For this purpose, we assume a type `IDSupply` with operations

```
initSupply  :: IO IDSupply
thisID      :: IDSupply → ID
leftSupply  :: IDSupply → IDSupply
rightSupply :: IDSupply → IDSupply
```

and add a new argument of type `IDSupply` to each operation of the source program, i.e., a Curry operation of type

```
f :: τ1 → ··· → τn → τ
```

is translated into a Haskell function of type

```
f :: τ1 → ··· → τn → IDSupply → τ
```

Conceptually, one can consider `IDSupply` as an infinite set of identifiers that is created at the beginning of an evaluation by the operation `initSupply`. The operation `thisID` takes some identifier from this set, and `leftSupply` and `rightSupply` split this set into two disjoint subsets without the identifier obtained by `thisID`. The split operations `leftSupply` and `rightSupply` are used when an operation calls two[4] other operations in the right-hand side of a rule. In this case, the called operations must be supplied with their individual disjoint identifier supplies. For instance, the operation `main` defined by

```
main :: Bool
main = xorSelf aBool
```

is translated into

```
main :: IDSupply → Bool
main s = xorSelf (aBool (leftSupply s)) (rightSupply s)
```

Any choice in the right-hand side of a rule gets its own identifier by the operation `thisID`, as in

```
aBool s = Choice (thisID s) True False
```

The type `IDSupply` can be implemented in various ways. The simplest implementation uses unbounded integers:

```
type IDSupply = Integer
initSupply    = return 1
thisID      n = n
leftSupply  n = 2*n
rightSupply n = 2*n+1
```

There are other more sophisticated implementations available [10]. Actually, our compilation system is parameterized over different implementations of `IDSupply` in order to perform some experiments and choose the most appropiate for a given application. Each implementation must ensure that, if s is a value of type `IDSupply`, then `thisID`$(o_1(\ldots(o_n\ s)\ldots))$ and `thisID`$(o'_1(\ldots(o'_m\ s)\ldots))$ are different identifiers provided that $o_i, o'_j \in \{\texttt{leftSupply}, \texttt{rightSupply}\}$ and $o_1 \cdots o_n \neq o'_1 \cdots o'_m$.

3.2 The Basic Translation Scheme

Functional logic computations can also fail, e.g., due to partially defined operations. Computing with failures is a typical programming technique and provides for specific programming patterns [6]. Hence, in contrast to functional programming, a failing computation should not abort the complete evaluation but it should be considered as some part of a computation that does not produce a meaningful result. In order to implement this behavior, we extend each data type by a further constructor `Fail` and complete

[4] The extension to more than two is straightforward.

each operation containing matching rules by a final rule that matches everything and returns the value `Fail`. For instance, consider the definition of lists

```
data List a = Nil | Cons a (List a)
```

and an operation to extract the first element of a non-empty list:

```
head :: List a → a
head (Cons x xs) = x
```

The type definition is extended as follows:[5]

```
data List a = Nil | Cons a (List a) | Choice (List a) (List a) | Fail
```

The operation `head` is extended by an identifier supply and further matching rules:

```
head :: List a → IDSupply → a
head (Cons x xs)      s = x
head (Choice i x1 x2) s = Choice i (head x1 s) (head x2 s)
head _                s = Fail
```

Note that the final rule returns `Fail` if `head` is applied to the empty list as well as if the matching argument is already a failed computation, i.e., it also propagates failures.

As already discussed above, an occurrence of "`?`" in the right-hand side is translated into a `Choice` supplied with a fresh identifier by the operation `thisID`. In order to ensure that each occurrence of "`?`" in the source program get its own identifier, all choices and all operations in the right-hand side of a rule get their own identifier supplies via appropriate applications of `leftSupply` and `rightSupply` to the supply of the defined operation. For instance, a rule like

```
main2 = xor aBool (False ? True)
```

is translated into

```
main2 s = let s1 = leftSupply  s
              s2 = rightSupply s
              s3 = leftSupply  s2
              s4 = rightSupply s2
          in xor (aBool s3) (Choice (thisID s4) False True) s1
```

An obvious optimization, performed by our compiler, is a *determinism analysis*. If an operation does not call, neither directly nor indirectly through other operations, the choice operation "`?`", then it is not necessary to pass a supply for identifiers. In this case, the `IDSupply` argument can be omitted so that the generated code is nearly identical to a corresponding functional program (apart from the additional rules to match the constructors `Choice` and `Fail`).

As mentioned in Section 2, our compiler translates occurrences of logic variables into generators. Since these generators are standard non-deterministic operations, they are translated like any other operation. For instance, the operation `aBool` is a generator for Boolean values and its translation into Haskell has been presented above.

[5] Actually, our compiler performs some renamings to avoid conflicts with predefined Haskell entities and introduces type classes to resolve overloaded symbols like `Choice` and `Fail`.

A more detailed discussion of this translation scheme can be found in the original proposal [12]. The correctness of this transformation from non-deterministic source programs into deterministic target programs is formally shown in [11].

3.3 Extracting Values

So far, our generated operations compute all the non-deterministic values of an expression represented by a structure containing `Choice` constructors. In order to extract the various values from this structure, we have to define operations that compute all possible choices in some order where the choice identifiers are taken into account. To provide a common interface for such operations, we introduce a data type to represent the general outcome of a computation,

```
data Try a = Val a | Choice ID a a | Fail
```

together with an auxiliary operation:[6]

```
try :: a  → Try a
try (Choice i x y) = Choice i x y
try Fail           = Fail
try x              = Val x
```

In order to take the identity of choices into account when extracting values, one has to remember which choice (e.g., left or right branch) has been made for some particular choice. Therefore, we introduce the type

```
data Choice = NoChoice | ChooseLeft | ChooseRight
```

where `NoChoice` represents the fact that a choice has not yet been made. Furthermore, we need operations to lookup the current choice for a given identifier or change its choice:

```
lookupChoice :: ID  → IO Choice
setChoice    :: ID  → Choice  → IO ()
```

In Haskell, there are different possibilities to implement a mapping from choice identifiers to some value of type `Choice`. Our implementation supports various options together with different implementations of `IDSupply`. For instance, a simple but efficient implementation can be obtained by using updatable values, i.e., the Haskell type `IORef`. In this case, choice identifiers are memory cells instead of integers:

```
newtype ID = ID (IORef Choice)
```

Consequently, the implementation of `IDSupply` requires an infinite set of memory cells which can represented as a tree structure:

```
data IDSupply = IDSupply ID IDSupply IDSupply
thisID      (IDSupply r _ _) = r
leftSupply  (IDSupply _ s _) = s
rightSupply (IDSupply _ _ s) = s
```

[6] Note that the operation `try` is not really polymorphic but overloaded for each data type and, therefore, defined in instances of some type class.

The infinite tree of memory cells (with initial value `NoChoice`) can be constructed as follows, where `unsafeInterleaveIO` is used to construct the tree on demand:

```
initSupply = getIDTree

getIDTree = do s1 <- unsafeInterleaveIO getIDTree
               s2 <- unsafeInterleaveIO getIDTree
               r  <- unsafeInterleaveIO (newIORef NoChoice)
               return (IDSupply (ID r) s1 s2)
```

Using memory cells, the implementation of the lookup and set operations is straightforward:

```
lookupChoice (ID ref) = readIORef ref
setChoice (ID ref) c = writeIORef ref c
```

Now we can print all values contained in a choice structure in a depth-first manner by the following operation:

```
printValsDFS :: Try a  → IO ()
printValsDFS (Val v)          = print v
printValsDFS Fail             = return ()
printValsDFS (Choice i x1 x2) = lookupChoice i >>= choose
 where
  choose ChooseLeft  = printValsDFS (try x1)
  choose ChooseRight = printValsDFS (try x2)
  choose NoChoice    = do newChoice ChooseLeft  x1
                          newChoice ChooseRight x2

  newChoice ch x = do setChoice i ch
                      printValsDFS (try x)
                      setChoice i NoChoice
```

This operation prints a computed value and ignores failures. If there is some choice, it checks whether a choice for this identifier has already been made (note that the initial value for all identifiers is `NoChoice`). If a choice has been made, it follows this choice. Otherwise, the left choice is made and stored. After printing all the values w.r.t. this choice, the choice is undone (like in backtracking) and the right choice is made and stored.

For instance, to print all values of the expression `main` defined in Section 3.1, we evaluate the Haskell expression

```
initSupply >>= \s  → printValsDFS (try (main s))
```

Thus, we obtain the output

```
False
False
```

In general, one has to propagate all choices and failures to the top level of a computation before printing the results. Otherwise, the operation `try` applied to an expression like "`Just aBool`" would return a `Val`-structure instead of a `Choice` so that the main operation `printValsDFS` would miss the non-determinism of the result value. Therefore, we have to compute the normal form of the main expression before passing it to the operation `try`. Hence, the result values of `main` are printed by evaluating

```
initSupply >>= \s  → printValsDFS (try (id $!! main s))
```

where "f `$!!` x" denotes the application of the operation f to the normal form of its argument x. This has the effect that a choice or failure occurring somewhere in a computation will be moved (by the operation "`$!!`") to the root of the main expression so that the corresponding search strategy can process it. This ensures that, after the computation to a normal form, an expression without a `Choice` or `Fail` at the root is a value, i.e., it does not contain a `Choice` or `Fail`.

Of course, printing all values via depth-first search is only one option which is not sufficient in case of infinite search spaces. For instance, one can easily define an operation that prints only the first solution. Due to the lazy evaluation strategy of Haskell, such an operation can also be applied to infinite choice structures. In order to abstract from these different printing options, our implementation contains a more general approach by translating choice structures into monadic structures w.r.t. various strategies (depth-first search, breadth-first search, iterative deepening, parallel search). This allows for an independent processing of the resulting monadic structures, e.g., by an interactive loop where the user can request the individual values.

4 Benchmarks

In this section we evaluate our compiler by comparing the efficiency of the generated Haskell programs to various other systems, in particular, other implementations of Curry. For our comparison with other Curry implementations, we consider PAKCS [25] (Version 1.9.2) which compiles Curry into Prolog [5] (based on SICStus-Prolog 4.1.2, a SWI-Prolog 5.10 back end is also available but much slower). PAKCS has been used for a number of practical applications of Curry. Another mature implementation we consider is MCC [32] (Version 0.9.10) which compiles Curry into C. MonC [13] is a compiler from Curry into Haskell. It is based on a monadic representation of non-deterministic computations where sharing is explicitly managed by the technique proposed in [19]. Since this compiler is in an experimental state, we could not execute all benchmarks with MonC (these are marked by "n/a").

The functional logic language TOY [30] has many similarities to Curry and the TOY system compiles TOY programs into Prolog programs. However, we have not included a comparison in this paper since [5] contains benchmarks showing that the implementation of sharing used in PAKCS produces more efficient programs.

Our compiler has been executed with the Glasgow Haskell Compiler (GHC 6.12.3, option -O2). All benchmarks were executed on a Linux machine running Debian 5.0.7 with an Intel Core 2 Duo (3.0GHz) processor. The timings were performed with the time command measuring the execution time (in seconds) of a compiled executable for each benchmark as a mean of three runs. "oom" denotes a memory overflow in a computation.

The first collection of benchmarks[7] (Fig. 2) are purely first-order functional programs. The Prolog (SICStus, SWI) and Haskell (GHC) programs have been rewritten

[7] All benchmarks are available at `http://www-ps.informatik.uni-kiel.de/kics2/benchmarks/`.

System	ReverseUser	Reverse	Tak	TakPeano
KiCS2	0.12	0.12	0.21	0.79
PAKCS	2.05	1.88	39.80	62.43
MCC	0.43	0.47	1.21	5.49
MonC	23.39	22.00	20.37	oom
GHC	0.12	0.12	0.04	0.49
SICStus	0.39	0.29	0.49	5.20
SWI	1.63	1.39	1.84	11.66

Fig. 2. Benchmarks: first-order functional programs

according to the Curry formulation. "ReverseUser" is the naive reverse program applied to a list of 4096 elements, where all data (lists, numbers) are user-defined. "Reverse" is the same but with built-in lists. "Tak" is a highly recursive function on naturals [34] applied to arguments (27,16,8) and "TakPeano" is the same but with user-defined natural numbers in Peano representation. Note that the Prolog programs use a strict evaluation strategy in contrast to all others. Thus, the difference between PAKCS and SICStus shows the overhead to implement lazy evaluation in Prolog.

One can deduce from these results that one of the initial goals for this compiler is satisfied, since functional Curry programs are executed almost with the same speed as their Haskell equivalents. An overhead is visible if one uses built-in numbers (due to the potential non-deterministic values, KiCS2 cannot directly map operations on numbers into the Haskell primitives) where GHC can apply specific optimizations.

System	ReverseHO	Primes	PrimesPeano	Queens	QueensUser
KiCS2	oom	1.22	0.30	10.02	13.08
KiCS2HO	0.24	0.09	0.27	0.65	0.73
PAKCS	7.97	14.52	23.08	81.72	81.98
MCC	0.27	0.32	1.77	3.25	3.62
MonC	oom	16.74	oom	oom	oom
GHC	0.24	0.06	0.22	0.06	0.11

Fig. 3. Benchmarks: higher-order functional programs

The next collection of benchmarks (Fig. 3) considers higher-order functional programs so that we drop the comparison to first-order Prolog systems. "ReverseHO" reverses a list with one million elements in linear time using higher-order functions like `foldl` and `flip`. "Primes" computes the 2000th prime number via the sieve of Eratosthenes using higher-order functions, and "PrimesPeano" computes the 256th prime number but with Peano numbers and user-defined lists. Finally, "Queens" (and "QueensUser" with user-defined lists) computes the number of safe positions of 11 queens on a 11×11 chess board.

As discussed above, our compiler performs an optimization when all operations are deterministic. However, in the case of higher-order functions, this determinism optimization cannot be performed since any operation, i.e., also a non-deterministic opera-

tion, can be passed as an argument. As shown in the first line of this table, this considerably reduces the overall performance. To improve this situation, our compiler generates two versions of a higher-order function: a general version applicable to any argument and a specialized version where all higher-order arguments are assumed to be deterministic operations. Moreover, we implemented a program analysis to approximate those operations that call higher-order functions with deterministic operations so that their specialized versions are used. The result of this improvement is shown as "KiCS2HO" and demonstrates its usefulness. Therefore, it is always used in the subsequent benchmarks.

System	PermSort	PermSortPeano	Last	RegExp
KiCS2HO	2.83	3.68	0.14	0.49
PAKCS	26.96	67.11	2.61	12.70
MCC	1.46	5.74	0.09	0.57
MonC	48.15	916.61	n/a	n/a

Fig. 4. Benchmarks: non-deterministic functional logic programs

To evaluate the efficiency of non-deterministic computations (Fig. 4), we sort a list containing 15 elements by enumerating all permutations and selecting the sorted ones ("PermSort" and "PermSortPeano" for Peano numbers), compute the last element `x` of a list `xs` containing 100,000 elements by solving the equation "`ys++[x] =:= xs`" (the implementation of unification and variable bindings require some additional machinery that is sketched in Section 5.3), and match a regular expression in a string of length 200,000 following the non-deterministic specification of `grep` shown in [8]. The results show that our high-level implementation is not far from the efficiency of MCC, and it is superior to PAKCS which exploits Prolog features like backtracking, logic variables and unification for these benchmarks.

Since our implementation represents non-deterministic values as Haskell data structures, we get, in contrast to most other implementations of Curry, one interesting improvement for free: deterministic subcomputations are shared even if they occur in different non-deterministic computations. To show this effect of our implementation, consider the non-deterministic sort operation `psort` (permutation sort) and the infinite list of all prime numbers `primes`, as used in the previous benchmarks, and the following definitions:

```
goal1 = [primes!!1003, primes!!1002, primes!!1001, primes!!1000]
goal2 = psort [7949,7937,7933,7927]
goal3 = psort [primes!!1003, primes!!1002, primes!!1001, primes!!1000]
```

In principle, one would expect that the sum of the execution times of `goal1` and `goal2` is equal to the time to execute `goal3`. However, implementations based on backtracking evaluate the primes occurring in `goal3` multiple times, as can be seen by the run times for PAKCS and MCC shown in Fig 5.

System	`goal1`	`goal2`	`goal3`
KiCS2HO	0.34	0.00	0.34
PAKCS	14.90	0.02	153.65
MCC	0.33	0.00	3.46

Fig. 5. Benchmarks: sharing over non-determinism

5 Further Features

In this section we sketch some additional features of our implementation. Due to lack of space, we cannot discuss them in detail.

5.1 Search Strategies

Due to the fact that we represent non-deterministic results in a data structure rather than as a computation as in implementations based on backtracking, we can provide different methods to explore the search space containing the different result values. We have already seen in Section 3.3 how this search space can be explored to print all values in depth-first order. Apart from this simple approach, our implementation contains various strategies (depth-first, breadth-first, iterative deepening, parallel search) to transform a choice structure into a list of results that can be printed in different ways (e.g., all solutions, only the first solution, or one after another by user requests). Actually, the user can set options to select the search strategy and the printing method.

Method	PermSort	PermSortPeano	NDNums
`printValsDFS`	2.82	3.66	∞
depth-first search	5.33	6.09	∞
breadth-first search	26.16	29.25	34.00
iterative deepening	9.16	10.91	0.16

Fig. 6. Benchmarks: comparing different search strategies

In order to compare the various search strategies, Fig. 6 contains some corresponding benchmarks. "PermSort" and "PermSortPeano" are the programs discussed in Fig. 4 and "NDNums" is the program

```
f n = f (n+1) ? n
```

where we look for the first solution of "`f 0 == 25000`" (obviously, depth-first search strategies do not terminate on this equation). All strategies except for the "direct print" method `printValsDFS` translate choice structures into monadic list structures in order to print them according to the user options. The benchmarks show that the overhead of this transformation is acceptable so that this more flexible approach is the default one.

We also made initial benchmarks with a parallel strategy where non-deterministic choices are explored via GHC's `par` construct. For the permutation sort we obtained a speedup of 1.7 when executing the same program on two processors, but no essential

speedup is obtained for more than two processors. Better results require a careful analysis of the synchronization caused by the global structure to manage the state of choices. This is a topic for future work.

5.2 Encapsulated Search

In addition to the different search strategies to evaluate top-level expressions, our system also contains a primitive operation

```
searchTree :: a → IO (SearchTree a)
```

to translate the search space caused by the evaluation of its argument into a tree structure of the form

```
data SearchTree a = Value a | Fail | Or (SearchTree a) (SearchTree a)
```

With this primitive, the programmer can define its own search strategy or collect all non-deterministic values into a list structure for further processing [15].

5.3 Logic Variables and Unification

Although our implementation is based on eliminating all logic variables from the source program by introducing generators, many functional logic programs contain equational constraints (e.g., compare the example "Last" of Fig. 4) to put conditions on computed results. Solving such conditions by generating all values is not always a reasonable approach. For instance, if `xs` and `ys` are free variables of type `[Bool]`, the equational constraint "`xs=:=ys`" has an infinite number of solutions. Instead of enumerating all these solutions, it is preferable to delay this enumeration but remember the condition that both `xs` and `ys` must always be evaluated to the same value. This demands for extending the representation of non-deterministic values by the possibility to add equational constraints between different choice identifiers. Due to lack of space, we have to omit the detailed description of this extension. However, it should be noted that the examples "Last" and "RegExp" of Fig. 4 show that unification can be supported with a reasonable efficiency.

6 Conclusions and Related Work

We have presented a new system to compile functional logic programs into purely functional programs. In order to be consistent with the call-time choice semantics of functional logic languages like Curry or TOY, we represent non-deterministic values in choice structures where each choice has an identification. Values for such choice identifiers are passed through non-deterministic operations so that fresh identifiers are available when a new choice needs to be created. The theoretical justification of this implementation technique is provided in [11]. Apart from the parser, where we reused an existing one implemented in Haskell, the compiler is completely written in Curry.

Due to the representation of non-deterministic values as data, our system easily supports various search strategies in constrast to Prolog-based implementations. Since we compile Curry programs into Haskell, we can exploit the implementation efforts

done for functional programming. Hence, purely functional parts of functional logic programs can be executed with almost the same efficiency as Haskell programs. Our benchmarks show that even the execution of the non-deterministic parts can compete with other implementations of Curry.

In the introduction we already discussed the various efforts to implement functional logic languages, like the construction of abstract machines [9,27,32] and the compilation into Prolog [5] or Haskell [13,15,16]. Our benchmarks show that an efficient implementation by compiling into a functional language depends on carefully handling the sharing of non-deterministic choices. For instance, our previous implementation [13], where sharing is explicitly managed by the monadic techniques proposed in [19], has not satisfied the expectations that came from the benchmarks reported in [19]. Due to these experiences, in our new compiler we use the compilation scheme initially proposed in [12] which produces much faster code, as shown in our benchmarks.

If non-deterministic results are collected in data structures, one has more fine-grained control over non-deterministic steps. For instance, [2] proposes pull-tab steps to move non-determinism from arguments to the result position of a function. Antoy [4] shows that single pull-tab steps are semantics-preserving. Thus, it is not necessary to move each choice to the root of an expression, as done in our implementation, but one could also perform further local computations in the arguments of a choice before moving it up. This might be a reasonable strategy if all non-deterministic values are required but many computations fail. However, the general effects of such refinements need further investigations.

Our implementation has many opportunities for optimization, like better program analyses to approximate purely deterministic computations. We can also exploit advanced developments in the implementation of Haskell, like the parallel evaluation of expressions. These are interesting topics for future work.

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. A. Alqaddoumi, S. Antoy, S. Fischer, and F. Reck. The pull-tab transformation. In *Proc. of the Third International Workshop on Graph Computation Models*, pages 127–132. Enschede, The Netherlands, 2010. Available at http://gcm-events.org/gcm2010/pages/gcm2010-preproceedings.pdf.
3. 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.
4. S. Antoy. On the correctness of the pull-tab transformation. In *To appear in Proceedings of the 27th International Conference on Logic Programming (ICLP 2011)*, 2011.
5. 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.
6. 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.

7. 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.
8. S. Antoy and M. Hanus. Functional logic programming. *Communications of the ACM*, 53(4):74–85, 2010.
9. 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.
10. L. Augustsson, M. Rittri, and D. Synek. On generating unique names. *Journal of Functional Programming*, 4(1):117–123, 1994.
11. B. Braßel. *Implementing Functional Logic Programs by Translation into Purely Functional Programs*. PhD thesis, Christian-Albrechts-Universität zu Kiel, 2011.
12. B. Braßel and S. Fischer. From functional logic programs to purely functional programs preserving laziness. In *Pre-Proceedings of the 20th Workshop on Implementation and Application of Functional Languages (IFL 2008)*, 2008.
13. B. Braßel, S. Fischer, M. Hanus, and F. Reck. Transforming functional logic programs into monadic functional programs. In *Proc. of the 19th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2010)*, pages 30–47. Springer LNCS 6559, 2011.
14. B. Braßel, M. Hanus, and M. Müller. High-level database programming in Curry. In *Proc. of the Tenth International Symposium on Practical Aspects of Declarative Languages (PADL'08)*, pages 316–332. Springer LNCS 4902, 2008.
15. B. Braßel and F. Huch. On a tighter integration of functional and logic programming. In *Proc. APLAS 2007*, pages 122–138. Springer LNCS 4807, 2007.
16. 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.
17. R. Caballero and F.J. López-Fraguas. A functional-logic perspective of parsing. In *Proc. 4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99)*, pages 85–99. Springer LNCS 1722, 1999.
18. S. Fischer. A functional logic database library. In *Proc. of the ACM SIGPLAN 2005 Workshop on Curry and Functional Logic Programming (WCFLP 2005)*, pages 54–59. ACM Press, 2005.
19. S. Fischer, O. Kiselyov, and C. Shan. Purely functional lazy non-deterministic programming. In *Proceeding of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP 2009)*, pages 11–22. ACM, 2009.
20. 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.
21. M. Hanus. A functional logic programming approach to graphical user interfaces. In *International Workshop on Practical Aspects of Declarative Languages (PADL'00)*, pages 47–62. Springer LNCS 1753, 2000.
22. M. Hanus. High-level server side web scripting in Curry. In *Proc. of the Third International Symposium on Practical Aspects of Declarative Languages (PADL'01)*, pages 76–92. Springer LNCS 1990, 2001.
23. M. Hanus. Type-oriented construction of web user interfaces. In *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'06)*, pages 27–38. ACM Press, 2006.
24. M. Hanus. Multi-paradigm declarative languages. In *Proceedings of the International Conference on Logic Programming (ICLP 2007)*, pages 45–75. Springer LNCS 4670, 2007.

25. 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/`, 2010.
26. M. Hanus and S. Koschnicke. An ER-based framework for declarative web programming. In *Proc. of the 12th International Symposium on Practical Aspects of Declarative Languages (PADL 2010)*, pages 201–216. Springer LNCS 5937, 2010.
27. 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.
28. M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.8.2). Available at `http://www.curry-language.org`, 2006.
29. H. Hussmann. Nondeterministic algebraic specifications and nonconfluent term rewriting. *Journal of Logic Programming*, 12:237–255, 1992.
30. F. López-Fraguas and J. Sánchez-Hernández. TOY: A multiparadigm declarative system. In *Proc. of RTA'99*, pages 244–247. Springer LNCS 1631, 1999.
31. F.J. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. A simple rewrite notion for call-time choice semantics. In *Proceedings of the 9th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'07)*, pages 197–208. ACM Press, 2007.
32. W. Lux. Implementing encapsulated search for a lazy functional logic language. In *Proc. 4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99)*, pages 100–113. Springer LNCS 1722, 1999.
33. J.J. Moreno-Navarro, H. Kuchen, R. Loogen, and M. Rodríguez-Artalejo. Lazy narrowing in a graph machine. In *Proc. Second International Conference on Algebraic and Logic Programming*, pages 298–317. Springer LNCS 463, 1990.
34. W. Partain. The nofib benchmark suite of Haskell programs. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, pages 195–202. Springer, 1993.
35. S. Peyton Jones, editor. *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press, 2003.
36. P. Wadler. How to replace failure by a list of successes. In *Functional Programming and Computer Architecture*, pages 113–128. Springer LNCS 201, 1985.
37. F. Zartmann. Denotational Abstract Interpretation of Functional Logic Programs. In *Proc. of the 4th International Symposium on Static Analysis (SAS'97)*, pp. 141–156. Springer LNCS 1302, 1997.