

The Kiel Curry System KiCS ^{*}

Bernd Braßel and Frank Huch
CAU Kiel, Germany
{bbr,fhu}@informatik.uni-kiel.de

Abstract. This paper presents the Kiel Curry System (KiCS) for the lazy functional logic language Curry. Its main features beyond other Curry implementations are: flexible search control by means of search trees, referentially transparent encapsulation and sharing across non-determinism.

1 Introduction

The lazy functional logic programming language Curry [?,?] combines the functional and the logical programming paradigms as seamlessly as possible. However, existing implementations of Curry, like PAKCS [?], the Münster Curry Compiler (MCC) [?] and the Sloth system [?], all contain seams which cannot easily be fixed within these existing systems.

To obtain a really seamless implementation, we developed a completely new implementation idea and translation scheme for Curry. The basic idea of the implementation is handling non-determinism (and free variables) by means of tree data structures for the internal representation of the search space. In all existing systems this search space is directly traversed by means of a special search strategy (usually depth-first search). This makes the integration of important features like user-defined search strategies, encapsulation and sharing across non-determinism almost impossible for these systems.

Since the logical features of Curry are now treated like data structures, the remaining part is more closely related to lazy functional programming (e.g. in Haskell [?]). As a consequence, we decided to use Haskell as target language in our new compiler in contrast to Prolog, as in PAKCS and Sloth, or C in MCC. The advantage of using a lazy functional target language is that most parts of the source language can be left unchanged (especially the deterministic ones) and the efficient machinery behind the Haskell compiler, e.g. the Glasgow Haskell Compiler (ghc), can be reused without large programming effort.

KiCS provides an interactive user interface in which arbitrary expressions over the program can be evaluated and a compiler for generating a binary executable for a `main` function of a program, similar to other existing systems. The next sections present the new key features of the Curry implementation KiCS, which allow seamless programming within Curry.

^{*} This work has been partially supported by the German Research Council (DFG) under grant Ha 2457/5-2.

2 Representing Search

To represent search in Haskell, our compiler employs the concept proposed in [?]. There each non-deterministic computation yields a data structure representing the actual search space in form of a tree. The definition of this representation is independent of the search strategy employed and is captured by the following algebraic data type:

```
data SearchTree a = Fail | Value a | Or [SearchTree a]
```

Thus, a non-deterministic computation yields either the successful computation of a completely evaluated term v (i.e., a term without defined functions) represented by `Value v`, an unsuccessful computation (`Fail`), or a branching to several subcomputations represented by `Or [t1, ..., tn]` where t_1, \dots, t_n are search trees representing the subcomputations.

The important point is that this structure is provided lazily, i.e., search trees are only evaluated to head normal form. By means of pattern matching on the search tree, a programmer can explore the structure and demand the evaluation of subtrees. Hence, it is possible to define arbitrary search strategies on the structure of search trees. Depth-first search can be defined as follows:

```
depthFirst :: SearchTree a -> [a]
depthFirst (Val v) = [v]
depthFirst Fail   = []
depthFirst (Or ts) = concatMap depthFirst ts
```

Evaluating the search tree lazily, this function evaluates the list of all values in a lazy manner, too. As an example for the definition of another search strategies, breadth-first search can be defined as follows:

```
breadthFirst :: SearchTree a -> [a]
breadthFirst st = unfoldOrs [st]
  where
    partition (Value x) y = let (vs,ors) = y in (x:vs,ors)
    partition (Or xs)    y = let (vs,ors) = y in (vs,xs++ors)
    partition Fail      y = y
    partition Suspend   y = y

    unfoldOrs [] = []
    unfoldOrs (x:xs) =
      let (vals,ors) = foldr partition ([],[]) (x:xs)
          in vals ++ unfoldOrs ors
```

The concept also allows to formulate a *fair search*. Fair search is realized by employing the multi-threading capabilities of current Haskell implementations. Therefore the definition of fair search is primitive from the point of the Curry system.

```
fairSearch :: SearchTree a -> IO [a]
fairSearch external
```

Search trees are obtained by *encapsulated search*. In [?] it is shown in many exam-

ples and considerations that the interactions between encapsulation and laziness is very complicated and prone to many problems. [?] also contains a wishlist for future implementations of encapsulation. KICS is the first implementation to fully meet this wish list. Concretely, there are two methods to obtain search trees:

1. The *current* (top-level) state of search can only be accessed via an io-action `getSearchTree :: a -> IO (SearchTree a)`.
2. A conceptual copy of a given term can be obtained by the primitive operation `searchTree :: a -> SearchTree a`. This copy does not share any of the non-deterministic choices of the main branch of computation, although all deterministic computations are shared, i.e., only computed once, cf. the next subsection.

This dual concept of encapsulation avoids all known conflicts with the other features of functional logic languages.

3 Sharing across Non-Determinism

Another key feature for a seamless integration of lazy functional and logic programming is sharing across non-determinism, as the following example shows:

Example 1 (Sharing across Non-Determinism). We consider parser combinators which can elegantly make use of the non-determinism of functional logic languages to implement the different rules of a grammar. A simple set of parser combinators can be defined as follows:

```

type Parser a = String -> (String,a)

succ :: a -> Parser a
succ r cs = (cs,r)

sym :: Char -> Parser Char
sym c (c':cs) | c==c' = (cs,c)

(<*>) :: Parser (a -> b) -> Parser a -> Parser b
(p1 <*> p2) str = case p1 str of
                    (str1,f) -> case p2 str1 of
                                (str2,x) -> (str2,f x)

(<$>) :: (a -> b) -> Parser a -> Parser b
f <$> p = succ f <*> p

parse :: Parser a -> String -> a
parse p str = case p str of
                ("",r) -> r

```

As an example for a non-deterministic parser, we construct a parser for the inherent ambiguous language of palindromes without marked center $L = \{w\bar{w} \mid$

$w \in \{a, b\}^1$, which, if parsing is possible, returns the word w and fails otherwise:

```
pal :: P String
pal = (\ c str _ -> c:str) <$> sym 'a' <*> p1 <*> sym 'a' ?
      (\ c str _ -> c:str) <$> sym 'b' <*> p1 <*> sym 'b' ?
succ ""
```

where $? :: a \rightarrow a \rightarrow a$ is the built-in operator for branching, defined as

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

In all Curry implementations the parser `p1` analyses a `String` of length 100 within milliseconds. We call this time t_{parse} .

Unfortunately, this program does not scale well with respect to the time it takes to compute the elements of the list to be parsed. Let's assume that the time to compute an element within the list $[e_1, \dots, e_{100}]$ is $t \gg t_{parse}$ and constructing the list $[e_1, \dots, e_{100}]$ takes time $100 \cdot t$. Then one would expect the total time to compute `parse pal [e1, ..., e100]` is $100 \cdot t + t_{parse}$. But measurements for the Curry implementation PAKCS ([?]) show that e.g., for $t = 0.131s$ it takes more than $5000 \cdot t$ to generate a solution for a palindrome $[e_1, \dots, e_{100}]$ and $9910 \cdot t$ to perform the whole search for all solutions. We obtain similar results for all the other existing implementations of lazy functional languages.

The reason is that all these systems do not provide sharing across non-determinism. For each non-deterministic branch in the parser the elements of the remaining list are computed again and again. Only values which are evaluated before non-determinism occurs are shared over the non-deterministic branches. This behavior is not only a matter of leaking implementations within these systems. There also exists no formal definition for sharing across non-determinism yet.

The consequence of this example is that programmers have to avoid using non-determinism, if a function might later be applied to expensive computations. I.e., non-determinism has to be avoided completely. Laziness even complicates this problem: many evaluations are suspended until their value is demanded. A programmer cannot know which values are already computed and, hence, may be used within a non-deterministic computation. Thus, sadly, when looking for possibilities to improve efficiency, the programmer in a lazy functional logic language is well advised to first and foremost try to eliminate the logic features he might have employed, which does not sound like a seamless integration of functional and logic programming.

KICS provides sharing across non-determinism which becomes possible by handling non-determinism by means of a tree data structure. There is only one global heap in which all non-deterministic computations are performed. The evaluation of a data structure in one non-deterministic branch is automatically shared to all other branches.

¹ We restrict to this small alphabet for simplicity of the example.

4 Narrowing instead of Residuation

Implementations of functional and functional logic languages usually provide numbers as an external data type and reuse the default implementation of the underlying language (e.g. C) for the implementation of operation like (+), (-), (<=), (=). This provides a very efficient implementation of pure computations within the high-level language.

However, in the context of functional logic languages, this approach results in a major drawback: numbers cannot be guessed by means of narrowing. As a consequence, complicated semantics extensions, like *residuation* [?,?], have been proposed which allow the user to use some restricted logical features in combination with numbers. The idea is that all (externally defined) functions on numbers suspend on free variables as long as their value is unbound. These residuated functions have to be combined with a generator which specifies all possible numbers of a free variable. As an example, we consider Pythagorean triples (a, b, c) with $a^2 + b^2 = c^2$. This problem can be implemented in the existing Curry implementations by means of the test and generate pattern [?]:

```
pyt | a*a+b*b:=c*c &
      c := member [1..] & b := member [1..c] & a := member [1..c]
      = (a,b,c)
      where a,b,c free

member (y:ys) = y ? member ys
```

First, the search tests whether a combination of a , b and c is a Pythagorean triple. Since in Curry natural numbers cannot be guessed this test is suspended by means of residuation for external functions (+) and (*). Now the programmer has to add good generator functions which efficiently enumerate the search space.

If, like common in Curry implementations, a depth first search strategy is used, it is especially important to restrict the search space for a and b to a finite domain. In practical applications, finding good generator functions is often much more complicated than in this simple example.

Moreover, residuation sacrifices completeness and detailed knowledge of internals is required to understand why the following very similar definition produces a run-time error:

```
pyt' | a*a + b*b := c*c = generate a b c where a,b,c free
generate a b c
  | c := member [1..] & b := member [1..c] & a := member [1..c]
  = (a,b,c)
```

On the other hand, the most beautiful standard examples for the expressive power of narrowing are defined for *Peano numbers*:

```

data Peano = 0 | S Peano

add 0      m = m
add (S n) m = S (add m n)

mult 0     _ = 0
mult (S n) m = add m (mult m n)

```

These functions cannot only be used for their intended purpose. We can also invert them to define new operations. For instance, the subtraction function can be defined by means of `add`:

```

sub n m | add r m := n = r
  where r free

```

Note, that we obtain a partial function, since there is no Peano number representation for negative numbers. By means of narrowing, a solution for the constraint `add r m := n` generates a binding for `r`. This binding is the result of `sub`.

For a solution of the Pythagorean triples, it is much easier to use Peano numbers instead of predefined integers. We can easily define a solution to this problem as follows:

```

pyt | (a 'mult' a) 'add' (b 'mult' b) := (c 'mult' c) = (a,b,c)
  where a,b,c free

```

It is not necessary to define any generator functions at all. In combination with breadth first search, all solutions are generated.

Furthermore, if the result `c` of the Pythagorean equation is already known (e.g., `c = 20`) and you are only interested in computing `a` and `b`, then it is even possible to compute this information with depth first search. The given equation already restricts the search space to a finite domain.

```

pyt | c := intToPeano 20 &
  (a 'mult' a) 'add' (b 'mult' b) := (c 'mult' c) = (a,b,c)
  where a,b,c free

```

We obtain the solutions: $(a, b) \in \{(0, 20), (12, 16), (16, 12), (20, 0)\}$.

Unfortunately, using Peano numbers is not appropriate for practical applications, as a simple computation using Peano numbers in PAKCS shows: computing the square of 1000 already takes 7 seconds. As a consequence, the developers of Curry [?] proposed the external type `Int` in combination with residuating, external functions for numbers.

In KICS numbers are implemented as binary encodings of natural numbers

```

data Nat = IHi | 0 Nat | I Nat

```

which in a second step are extended with an algebraic sign and a zero to represent arbitrary integers.

```

data Int = Pos Nat | Zero | Neg Nat

```

The implementation of the standard operations for numbers is presented in [?].

To get an impression, how the implementation works, we present the definition of the addition for natural numbers here:

```

add :: Nat -> Nat -> Nat
IHi 'add' m = succ m           1 + m = m + 1
O n 'add' IHi = I n           2n + 1 = 2n + 1
O n 'add' O m = O (n 'add' m) 2n + 2m = 2 · (n + m)
O n 'add' I m = I (n 'add' m) 2n + (2m + 1) = 2 · (n + m) + 1
I n 'add' IHi = O (succ n)    (2n + 1) + 1 = 2 · (n + 1)
I n 'add' O m = I (n 'add' m) (2n + 1) + 2m = 2 · (n + m) + 1
I n 'add' I m = O (succ n 'add' y) (2n + 1) + (2m + 1) = 2 · (n + 1 + m)

```

where `succ` is the successor function for `Nat`.

Using these numbers within `KICS`, many search problems can be expressed as elegantly as using Peano numbers. For instance, all solutions for the Pythagorean triples problem can in `KICS` (in combination with breadth first search) be computed with the following expression:

```
let a,b,c free in a*a + b*b := c*c &> (a,b,c)
```

For a fixed `c`, e.g. beforehand bound by `c := 20`, all solutions are also computed by depth first search.

The same implementation is used for characters in `KICS`, which allows to guess small strings as well. To avoid the overhead related to encoding characters as binary numbers, internally a standard representation as Haskell character is used in the case when characters are not guessed by narrowing.

5 Performance

Although we did not investigate much time in optimizing our compiler, performance is promising, as the following benchmarks show:

- naive: Naive reverse of a list with 10000 elements.
- fib: Fibonacci of 25.
- XML: Read and parse an XML document of almost 2MB.
- perm: Non-deterministically compute all 40320 permutations of the list `[1..8]`.

	PAKCS	MCC	KICS
naive:	22.50 s	4.24 s	3.89 s
fib:	3.33 s	0.06 s	0.30 s
XML:	-	3.55 s	8.75 s

Loading the large XML file is not possible in `PAKCS`. The system crashes, since memory does not suffice.

When comparing run-times for `fib`, one has to consider that `KICS` uses algebraic representation of numbers. Anyhow, we are faster than `PAKCS`, but much slower than the `MCC`. Here `MCC` is able to optimize a lot, but number cannot be guessed in `MCC`.

A comparison for non-deterministic computations is difficult. For several examples, KICS is much slower than PAKCS and MCC, which results from the additional overhead to construct the search trees data structures. On the other hand, in many examples, KICS is much slower than PAKCS and MCC, because of sharing across non-determinism. We think, that constructing fair examples here is almost impossible and hence we omit such examples here.

The implementation of KICS just started and there should be a lot of space for optimizations, which we want to investigate for future work. However, we think our approach is promising and it is the first platform for implementing new features for lazy functional-logic languages, like sharing across non-determinism and encapsulated search.