Seminar Report

Curry - a Combination of Functional and Logic Programming

Paul Borek
Paul.Borek@student.uibk.ac.at

17 February 2011

Supervisor: DI Andreas Schnabl

Abstract

Declarative languages have become more and more important. This observation holds not only for the academic world, but also in industry. Functional languages as Haskell, OCaml, Lisp or Python are very well known in certain areas of modern computer science such as agents programming, neuronal networks and automated theorem proving, and Logic languages as Prolog, ALF, Castor and Alloy are located in databases, artificial intelligence, natural language processing and many more. Due to this knowledge, combining these two paradigms could be a good thing: with lazy evaluation, higher order functions and monadic data structures from the functional paradigm and non-deterministic evaluation strategies, logical variables and rule based information modelling from the logic paradigm, a powerful language could be developed. These thoughts had also Michael Hanus, Sergio Antoy and the team around them, which modelled the theory and later the practical elaboration of the programming language Curry, which is described in this report.

Contents

1 Introduction **1**

2 Common Features **1**
- 2.1 Data Types 1
  - 2.1.1 Constraints 2
- 2.2 Predefined Operations 2
- 2.3 Functions 3

3 Functional Logic Features **4**
- 3.1 Logical Variables 5
- 3.2 Evaluation strategy 7
  - 3.2.1 Non-deterministic Functions 9

4 Practical Examples **10**
- 4.1 Regular Expressions 10
- 4.2 Merge-Sort 11

5 Actual Projects **12**

6 Conclusion **12**

Bibliography **15**

1 Introduction

This seminar report describes the programming language *Curry*, which takes its name from Haskell Brooks Curry [7], an American mathematician and logician (1900 - 1982), and was developed at the University of Kiel. It was introduced in 1995 by Michael Hanus and Sergio Antoy [3]. The background of the further development of Curry was that declarative languages (functional languages, logic languages, query languages ...) have a clear correspondence to mathematical logic, i.e., the programmer should only see *what* computation should be performed, not *how* the computation is done. Therefore programs are smaller and without side effects. Curry takes the two most important declarative paradigms (functional and logic programming) and combines them to a new programming language.

Compared to pure functional languages, as Haskell [4], functional logic languages have more expressive power due to the use of logical variables and built-in search mechanisms. On the other side compared to pure logical languages, as Prolog [6], functional logic languages have more efficient evaluation mechanisms (lazy evaluation) due to the (deterministic) reduction of functional expressions.

This report is divided into 3 sections. Section 2 reviews a selection of features which are covered in other languages and are especially important for the declarative paradigm. Section 3 describes aspects, which are either not typical for functional languages or build a bridge between the two fore cited paradigms. Section 4 then describes two examples written in Curry. After describing some bigger projects which are written in Curry, the last section gives a conclusion of the topic.

2 Common Features

The syntax of Curry borrows heavily from that of Haskell [4]. Curry introduces a single syntactic extension (the declaration of free variables) and changes the underlying evaluation strategy.

Thus, the structure of a Curry program does not differ much from the definition of a functional program:

Definition 2.1 (Curry program)**.** A Curry program is a set of definitions of data types and operations on values of these types (functions).

The next subsection introduces the data types in Curry.

2.1 Data Types

A *data type* is declared by enumerating all of its constructors with the respective argument types [1]:

```
data Bool = True | False
data BTree a = Leaf a
| Branch (BTree a) (BTree a)
```

Type	Declaration	Examples
Integer	`Int`	`...,-2,-1,0,1,2,...`
Float	`Float`	`...,3.1415, -0.65,...`
Boolean	`Bool`	`False, True`
Character	`Char`	`'a', 'b', 'c', ... '\n'`
String	`String`	`"hello", "world\n"`
Unit	`()`	`()`
List of τ	`[`τ`]`	`[], [1,2,3], 0:1:2:[]`
Tuple of τ_1, ..., τ_n	`(`τ_1`, ..., `τ_n`)`	`('a', 12), ("hi", 15), ...`
Success	`Success`	`success`

Table 1: Some of the built in types of Curry.

`BTree a` and `Bool` are *type constructors* (moreover `BTree` is a polymorphic type constructor with type variable `a`), while `True`, `False`, `Leaf` or `Branch (BTree a) (BTree a)` are *data constructors*. Like other languages, Curry has several built in types (some samples are given in Table 1).

Beside the standard built-in types, which are not different from other languages, `Success` is an interesting type: It only has the value `success` and it can be interpreted as the type of successful evaluations. Expressions of the type `Success` are called *constraints*.

2.1.1 Constraints

A *constraint* has, compared to normal expressions, a special role in Curry programs. Constraints build the bridge between functional and logic programming.

In Curry, each expression which returns `success` is a constraint. They can either explicitly return `success` or they can be built by applying specific operations between other constraints at the body of them. Such operations are explained in the next subsection. This specific return type leads to the use of a different evaluation strategy, which can bind values to free variables (also see Section 3).

Moreover, constraints in Curry are quite similar to queries in Prolog: while the evaluation of queries finds *nodes* of the SLD-tree and returns them, constraints can be seen as *branches* in the the SLD-tree. The type `Success` denotes then the "successful evaluation" of such a constraint.

Note that `Success` is something totally different from `Bool`, since `Success` does not have something like `False`. If the evaluation of a constraint can not be successful evaluated, the underlying evaluation strategy of Curry searches for other constraints, which instead are successfully evaluated (narrowing, see Section 3.2).

The next section introduces the *equational constraint*, which is used to construct constraints from other expressions by allowing only `success` if both sides are evaluable to unifiable data terms.

2.2 Predefined Operations

Some of the built-in operations are introduced in Table 2.

Description	**Identifier**	**Type**
Boolean equality	`==`	`a -> a -> Bool`
Boolean conjunction	`&&`	`Bool -> Bool -> Bool`
Boolean disjunction	`||`	`Bool -> Bool -> Bool`
Parallel conjunction	`&`	`Success -> Success -> Success`
Constrained equality	`=:=`	`a -> a -> Success`
Constrained expression	`&>`	`Success -> a -> a`

Table 2: Some of the built in operations of Curry.

Boolean equality, *Boolean conjunction* and *Boolean disjunction* are commonly used in other languages, i.e., like the corresponding operators at other languages as C, Java or OCaml, LISP and no longer explained in detail.

So, the first new operator is *constrained equality*, which is similar to Boolean equality except that constraints are evaluated. Therefore the return type is `Success` (Section 2.1.1). This type of equality checks whether both sides of the equality are evaluable to unifiable data terms (Section 3).

Parallel conjunction (`a & b`) denotes the concurrent evaluation of two constraints and returns `Success`, if both can be evaluated to `success` and if the evaluation of both sides implies a conjoint variable binding. Moreover, if `a` suspends the computation, the evaluation of `b` is proceeded which may cause the reactivation of `a` at some later time again. The correct synchronization of shared data is granted.

Constrained expression (`a &> b`) is an operation, which solves first constraint `a` and then evaluates `b`, where `b` can be an arbitrary expression (then the return type is that of `b`). Since a constraint is also an expression, this operation works also with two constraints (then `Success` is returned).

2.3 Functions

Since we are talking about a functional language, we need functions.

The definition of a function in functional languages is that it takes *one* argument and returns *one* value (`a -> b`). Here the arrow denotes the computation, which is performed by expressions where the argument is involved.

If we want functions with more than one argument (`a -> b -> c`), this is encoded always by a function with one argument, which returns another function. Thus, the associativity of function types is right (`a -> (b -> c)`). For example, consider the two following functions, where the (optionally) first line denotes the type signature:

```
square :: Int -> Int
square x = x * x

add :: Int -> Int -> Int
add a b = a + b
```

Higher order functions are a commonly used feature in functional languages, as Haskell or OCaml, i.e., functions which take other functions as argument, and are therefore also valid in Curry:

```
 my_map _ []      = []
 my_map f (x:xs)  = f x : map f xs
```

Like in functional languages, the evaluation is *lazy*, i.e., the computation of any expression is delayed until the expression's value is actually needed. This is used also in C or Java at the evaluation of Boolean expressions (short circuit).

Moreover, Curry allows the use of potentially infinite data structures. Consider the function:

```
from n = n : from (n+1)
```

The call `from 1` would lead to a memory overflow, since the result is infinite. However, with lazy evaluation, a call of `head (from 1)` would return 1 as expected (like in Haskell).

Also *pattern matching* is supported by Curry: functions are specified by different equations for different argument values. Hence, a function may be broken into more rules, depending on conditions by enumerating the different patterns explicitly. Beside the example of `my_map` which already used pattern matching, here another example:

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

Each rule defining a function can include one or more *conditions* of type `Bool` (*guards*):

```
max x y | x < y      = y
        | otherwise  = x
```

Here, `otherwise` is syntactic sugar for `True`.
But the guards in Curry differ from the ones in Haskell or OCaml: a constraint instead of a Boolean condition is allowed. In this case, this constraint is checked for satisfiability in order to apply the rule. Thus, the function call reduces to the right-hand side, only if the constraint is satisfied, otherwise it fails. Note that multiple conditions as above are not allowed for constraint conditions.

The next section introduces features which can be seen as a connection between functional and logic languages.

3 Functional Logic Features

Beside constraints, the type `Success` and operations on them (`=:=`, `&` and `&>`) almost everything was a typical instrument for functional languages until here. At the next two sections, we will see how these instruments can be used to model mechanisms of logic programming.

3.1 Logical Variables

Recall the definition of a logic program, which consists of *rules* and *facts*. Every rule has a *head* and a *body* except facts, which have an empty body. Consider the following Prolog program:

```
female(doris).
female(eva).
female(maria).

male(peter).

father(peter, doris).
father(peter, eva).

mother(maria, eva).

parent(X,Y) :- father(X,Y).
parent(X,Y) :- mother(X,Y).

daughter(X,Y) :- parent(Y,X), female(X).
```

Next, we can query a specific consequence by typing the query statement `?- daughter(X,peter).` to get all daughters of `peter`.

Prolog uses *unification* and *resolution* to get an answer, where unification is needed to find a suitable substitution (most general unifier) to find the most general answer for the query. Resolution takes a set of rules and the query. Then it selects an applicable rule and a goal and tries to unify them with the mgu. If possible, this mgu is returned. If not, there are no solutions and Prolog returns `False`. In other words, resolution and unification iterate over the result of applied rules of the program.

Since we heard that in Curry we can model constraints by writing down expressions with the return type `Success`, we can now define such functions and we may observe that these are expressions, which return always `success` and are therefore always successful.

For example by defining a rule `female Doris = success`, we model the information that the expression `female` applied on the value `Doris` (after declaring this data type with `data person = Doris`) is always `successful`. In a similar way, we model then the first seven rules of the recent logic program, after declaring the data type.

The only remaining rules are the last three: they all have non-empty bodies and have to be modeled with other rules. To combine multiple rules in Prolog, we have *disjunction* (by splitting rules into more or by using `;`) and *conjunction* (by using `,`).

Disjunction is modeled already in the Prolog program by splitting the rules into multiple ones, so we do the same in Curry. The evaluation strategy searches then for the first successfully evaluated constraint of the arguments and returns `success` if such a constraint could be found.

To model conjunction, recall the definition of parallel conjunction from Section 2.2. With this operation, we can now model conjunction of two constraints too.

To illustrate the previous example in Curry, consider the following program.

```
data person = Doris | Eva | Maria | Peter

female Doris = success
female Eva = success
female Maria = success

male Peter = success

father Peter Doris = success
father Peter Eva = success

mother Maria Eva = success

parent x y = father x y
parent x y = mother x y

daughter x y = parent y x & female x
```

However, by typing `daughter x Peter`, in Curry the interpreter answers with `ERROR: Expression contains unknown symbols: x`.

The reason is the following: other than in Prolog, not all variables occurring somewhere in the query are automatically existentially quantified. Thus *each free variable has to be declared explicitly.* This is done by adding the construct `where x free` at the end of the query statement. Instead of `x` more variables separated with commas are allowed here too.

At the end, the interpreter prints also the (first) answer:

```
Free variables in goal: x
Result: success
Bindings:
x=Doris
```

By typing again `y` or `a` (for all answers), also the second variable binding is printed and we are finished:

```
Result: success
Bindings:
x=Eva
No more solutions.
```

The next section introduces the underlying evaluation strategy, which differs from the standard evaluation method for functional languages (lazy evaluation).

3.2 Evaluation strategy

Consider the following example:

```
zs ++ [2] =:= [1,2] where zs free
```

This equation states that we are interested in suitable variable bindings for `zs` (where `++` denotes list concatenation).

An evaluation strategy based on term reduction would not work here, since we have to instantiate the free variable. This implies to deal with a mix between term reduction and variable introduction.

While functional programs return a data term at the end, functional logic languages deal with a pair, the so called *answer expression*:

Definition 3.1 (Answer expression)**.** An answer expression $a := \sigma \,[\!]\, e$ consists of a substitution σ and an expression e. We say that a is *solved* if e is a data term.

Usually more answer expressions can be computed during the evaluation, so they are combined through a *disjunction*, since one ore more answers may be possible:

$$\{\sigma_1 \,[\!]\, e_1 \mid \ldots \mid \sigma_n \,[\!]\, e_n\}$$

A single computation step performs a reduction in exactly one unsolved expression (for example from left to right in Prolog-like Curry implementations).

Example 3.2. Consider the following function definition:

```
f 0 = 2
f 1 = 3
```

A call of

```
f x where x free
```

would be evaluated to the following disjunction:

$$\{x = 0\} \,[\!]\, 2 \mid \{x = 1\} \,[\!]\, 3$$

In this simple example the result can be seen directly: if we would instantiate the variable by `0`, `2` would be the return value (first disjunct) and if we would instantiate it with `1`, we would get `3` as return value, which corresponds to the second disjunct.

Usually, functions can support one of the two following evaluation strategies:

Residuation Simply delay the evaluation of the free variables until they are bound.

Narrowing Perform an instantiation of the values of free variables, such that the program rules become applicable. To do this, the narrowing strategy determines the steps, which lead to a useful computation, i.e., it searches appropriate answer expressions to succeed the computation.

Note that narrowing is not a new approach. It is a well known strategy in automated theorem proving.

Operations which use residuation as evaluation strategy are called *rigid*. Such operations are arithmetic operations in general since variable guessing, as narrowing does, is not adequate for arithmetic operations.

Operations which use narrowing are called *flexible*.

To understand better the behavior of a narrowing mechanism, consider the following example:

Example 3.3. First of all, recall the standard definition of the append function on lists:

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

To perform the evaluation of `xs ++ [2] =:= [1,2] where xs free`, we first have to show the evaluation of the `=:=` operator, considering the Curry Report [5].

First of all, consider the following small example to show the (partially) computation of the constrained equality:

```
y : [2] =:= [1,2]
y : (2 : []) =:= 1 : (2 : [])
y =:= 1 & (2 : []) =:= (2 : [])
...
```

First of all, the syntactic sugar of list-representations are eliminated to understand the behavior of the operator.

The next step is then to split the equivalence into a conjunction of simpler constrained equations. In this case this is done by recursively applying the equivalence on both arguments of the list constructor.

By the definition of constrained equivalence, if one side is a data constructor (or in this case a number) and the other side a variable, we simply instantiate this variable with the corresponding value. This procedure is continued until all sides can evaluate to `success` (due to the specification of the operations `&` and `=:=`).

On the other side, such instantiations as seen in the third line of the previous listing form then the substitution-part of the answer expression.

Consider the following example (syntactic sugar was left inside):

$$\{xs = []\} \, [] \, \texttt{[2] =:= [1,2]} \mid \{xs = y : ys\} \, [] \, \texttt{y:(ys ++ [2]) =:= [1,2]}$$
$$\to^2 \{xs = 1 : ys\} \, [] \, \texttt{1:(ys ++ [2]) =:= [1,2]}$$
$$\to \{xs = 1 : ys, ys = []\} \, [] \, \texttt{1:[2] =:= [1,2]} \mid$$
$$\{xs = 1 : ys, ys = z : zs\} \, [] \, \texttt{1:(z:(zs++[2])) =:= [1,2]}$$
$$\to^2 \{xs = [1]\} \, [] \, \texttt{success}$$

The first disjunct is the substitution of `xs` by the empty list on the one side and by the list-constructor at the other side.

The empty list does not lead to `success`, because by having an expression `2:[] =:= 1:(2:[])`, the following reduction step would lead to a parallel conjunction `2 =:= 1 & [] =:= 2:[]`, where `2 =:= 1` fails. This answer expression is therefore removed from the disjunction in the first of the following two steps. Then the `y` is immediately instantiated to `1`, due to specification of `=:=` in the Curry Report [5].

At the next step the `ys` has to be instantiated. Again we have to instantiate all possible right-hand sides of the append-rules.

Since we assume an evaluation from left to right, we take the first disjunct in the last two steps and it succeeds immediately, so the binding in the substitution is simplified to `xs=[1]`.

3.2.1 Non-deterministic Functions

The evaluation strategy in the last subsection showed that we can instantiate values for free variables. Therefore we can write functions with different return values, since we allow answer expressions with different return values but equal bindings (the definition of the answer expression gives no restriction with respect to this attribute).

So, a function which returns one of its arguments can be written down:

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

This is the *choose*-operator, which is defined by the standard library of curry. An application of this operator could be the following (very inefficient, but demonstrative) sorting algorithm [5].

```
insert x ys     = x : ys
insert x (y:ys) = y : insert x ys
```

This function is non-deterministic, since both rules can be matched on a non-empty list.

By applying it on an arbitrary list, either the ordering of the first argument `x` is left, or swapped with the head of the second list (`y`). The most appropriate application for such a function is the permutation function:

```
perm [] = []
perm (x:xs) = insert x (perm xs)
```

After defining a function to permute a given list we now have to check whether a given list is sorted or not to complete this sorting algorithm:

```
sorted [] = []
sorted [x] = [x]
sorted (x:y:xs) | x<=y  = x : sorted (y:xs)
```

Now a call of `sorted (perm xs)` would return the sorted version of `xs`. The sorted property is checked here, and if it fails, Curry checks an alternative until the property holds. Moreover, Curry includes a demand-driven strategy, which does not evaluate the expression completely. For example the call of `sorted (3:1:perm ys)`, would avoid the remaining call of `perm ys`.

4 Practical Examples

To outline the practical part of Curry, I will show at first an example for representing *regular expressions*, where it can be seen that the combination of non-deterministic functions and pattern matching can be useful in a more concrete example (compared to the permutation-sort example in the last subsection) too. After this, the merge sort algorithm is implemented in Curry, where we can start the recursive calls of merge concurrently with the use of the parallel conjunction (from Section 2.2).

4.1 Regular Expressions

Consider a regular expression. Then in Curry we could define such an expression with:

```
data Reg_exp a = Empty_set | Bas a | Alt (Reg_exp a) (Reg_exp a)
                           | Conc (Reg_exp a) (Reg_exp a)
                           | Star (Reg_exp a)
```

Here `Bas` denotes a basic regular expression in the given alphabet (type) `a`, `Alt` is the choice between two regular expressions, `Conc` is the concatenation of two regular expressions and `Star` denotes zero or more repetitions of a regular expression. For example, the regular expression $a * b *$ would then be:

```
astarbstar = Conc (Star (Bas 'a')) (Star (Bas 'b')))
```

Additionally new operators on regular expressions, like the $+$ can be represented (which denotes one or more repetitions):

```
plus re = Conc re (Star re)
```

The next step is the definition of a function, which returns a word inside the language of a given regular expression. Here we can see the useful application of non-deterministic functions: the decision of the number of repetitions of `Star` and the choice of the regular expressions of `Alt` is left to the Curry interpreter:

```
sem (Empty_set) = []
sem (Bas a) = [a]
sem (Alt a b) = sem a ? sem b
sem (Conc a b) = sem a ++ sem b
sem (Star a) = [] ? sem (Conc a (Star a))
```

Now, by typing `sem re =:= word` the interpreter returns, if the given `word` is inside the language of the regular expression `re`.

To simulate `grep` from Unix, we could write the following line:

```
grep re s xs ys = xs ++ sem re ++ ys =:= s
```

This can be tested by typing for example:

```
 grep astarbstar "ab" xs ys where xs,ys free
```

which would print out the following substitutions:

xs	sem myregexp	ys	s
[]	[]	"ab"	"ab"
[]	"a"	"b"	"ab"
[]	"ab"	[]	"ab"
"a"	[]	"b"	"ab"
"a"	"b"	[]	"ab"
"ab"	[]	[]	"ab"

4.2 Merge-Sort

The next example is an implementation of the merge-sort algorithm [2]. Since `take` and `drop` are already defined in the `Prelude`, the functions for dividing a list into the first and the second half can be defined as follows:

```
firsthalf  xs = take (length xs ‘div‘ 2) xs
secondhalf xs = drop (length xs ‘div‘ 2) xs
```

Then we need the merge-part. Since we use `>`, this `merge` function is polymorphic and works for the same types for which `>` works:

```
merge [] ys zs = zs =:= ys
merge (x:xs) [] zs =  zs =:= x:xs
merge (x:xs) (y:ys) zs = if (x > y)
              then merge (x:xs) ys us & zs =:= y:us
              else merge xs (y:ys) vs & zs =:= x:vs
   where us,vs free
```

The first two rules catch instances, where the first or the second list is empty. In this cases the nonempty list is returned.

The third rule has then two nonempty lists as first argument and performs the real merge operation. Important to notice is the `&`, which in this case does not improve any parallelism, but grants the consistent computation of `us` and `vs` respectively. The constrained equality is then used to search for the right solution of `zs`. In other words, this implementation does not all the work, but delegates it to the evaluation strategy of `=:=` and `&`. Clearly, the existential quantification of `us` and `vs` has to be ensured.

Now we have to put these two steps together with the `sort`-function:

```
sort xs ys =
   if length xs < 2 then ys =:= xs
   else sort (firsthalf xs) us
        & sort (secondhalf xs) vs
        & merge us vs ys
   where us,vs free
```

The case where the given list has one or zero elements is caught by the then-branch of the if-then-else construct. If not, we recursively divide the list into two sub lists (`firsthalf` and `secondhalf`) and merge it concurrently. Due to the specification of the parallel conjunction, one side of the evaluation stops, if a value is demanded and then continued, if it is bound at the other side of the operation. The result is a sorted list.

5 Actual Projects

The Curry programming language was basically an experiment. Therefore, many tools *for* Curry are implemented *in* Curry (Debugging, IDEs, test environments, analysis environments).

Applications are counted among: they are to show, that such software can be written smarter in Curry and to test the Curry library exhaustively.

However, two exemplary applications and their thoughts are listed here, which are located in the CurryWiki with many others.[1]:

Spicey This software is the product of a master-project of Sven Koschnicke and supports the implementation of web-based systems by generating a initial implementation from an entity-relationship description of the underlying data. By reading this thesis one may notice, that the goal of the thesis was not only to write a software-development program, but to show the benefits of declarative descriptions inside the development of web-based applications in general. As major benefit he emphasize the simple extendability of functional and logic programs, namely by adding rules. To show this benefits in practice, he developed then the framework in Curry.

CurryWeb A system for the support of web-based learning. Shows in particular the practical use of several parts of the Curry library, for example the module `HTML`, which was developed to represent HTML tags with data types and supports the programmer with functions as useful abbreviations and the incorporation of CGI scripts with the `CgiRef` module.

As already mentioned: for further projects, visit the CurryWiki (`http://www-ps.informatik.uni-kiel.de/currywiki/applications`). Beside the listed projects on the Wiki, other nameable applications are not present.

6 Conclusion

As we saw in this report, the combination of logic and functional programming is possible. The benefits of both paradigms were extracted and merged into a programming language, which uses even concurrent mechanisms to gain a higher abstraction level.

[1] `http://www-ps.informatik.uni-kiel.de/currywiki/applications`

However, especially Section 3 showed that a significant knowledge is needed to understand the behavior of the compiler and the evaluation of such a language. Moreover, programming with two paradigms can be obfuscating, since the example of the logic program in Section 3.2 could be also modeled by using only functional components.

The real benefit is the use of non-deterministic functions, which showed a very short implementation of the `grep`-command of Unix. In purely functional languages, such a program would lead to a huge number of pattern matching rules and only with the exhaustive use of list-functions.

Nowadays, Curry has different applications. Anyway, these implementations are mostly implemented and kept at the University of Kiel, i.e., by Michael Hanus. By searching for papers concerning this topic, one may conclude that the boom of Curry is over now: last papers of Curry were from 2001.

Nevertheless, narrowing is still ongoing research, since it takes place at automated theorem proving, where it has an essential role.

Beside Curry, several approaches were developed to complete such a fusion too:

- Escher (mid-1990s, J.W. Lloyd)
- Oz (1991, Gerd Smolka)
- Babel (1988, J. J. Moreno-Navarro et. al.)
- LPG (1988, D. Bert et. al.)
- SLOG (1991, P. H. Cheong et. al.)

By comparing the years of occurrence, we can observe that this topic has reached the nuclear winter of research interest in computer science. However Curry is one of the latest attempts and is still used nowadays (even if rarely).

References

[1] S. Antoy and M. Hanus. Functional logic programming. *Commun. ACM*, 53(4):74–85, 2010.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition.* The MIT Press and McGraw-Hill Book Company, 2001.

[3] M. Hanus, H. Kuchen, and J. Moreno-Navarro. Curry: A truly functional logic language. In *Proc. ILPS'95 Workshop on Visions for the Future of Logic Programming*, pages 95–107, 1995.

[4] P. Hudak, S. L. P. Jones, P. Wadler, B. Boutel, J. Fairbairn, J. H. Fasel, M. M. Guzmán, K. Hammond, J. Hughes, T. Johnsson, R. B. Kieburtz, R. S. Nikhil, W. Partain, and J. Peterson. Report on the programming language haskell, a non-strict, purely functional language. *SIGPLAN Notices*, 27(5):1–, 1992.

[5] J. W. Lloyd. Programming in an Integrated Functional and Logic Language. *Journal of Functional and Logic Programming*, 1999(3), 1999.

[6] J. Wielemaker. SWI-Prolog, 1987. `http://www.swi-prolog.org/`.

[7] Wikimedia Foundation. Haskell Brooks Curry - Wikipedia, the free encyclopedia, 2011. `http://en.wikipedia.org/wiki/Haskell_Curry`.