FUNCTIONAL PEARL

Concurrent Distinct Choices∗

Sergio Antoy
Portland State University
Computer Science Department
P.O. Box 751, Portland, OR 97207, U.S.A.
antoy@cs.pdx.edu

Michael Hanus
Christian-Albrechts-Universität Kiel
Institut für Informatik
Olshausenstr. 40, D-24098 Kiel, Germany
mh@informatik.uni-kiel.de

Abstract

An injective finite mapping is an abstraction common to many programs. We describe the design of an injective finite mapping and its implementation in Curry, a functional logic language. Curry supports the concurrent asynchronous execution of distinct portions of a program. This condition prevents passing from one portion to another a structure containing a partially constructed mapping to ensure that a new choice does not violate the injectivity condition. We present some motivating problems and we show fragments of programs that solve these problems using our design and implementation.

1 Introduction

A finite mapping is one of the most common abstractions in computer programs. Many programming languages directly support this abstraction by offering a primitive type, the *array*, with a special notation to ease the implementation and use of finite mappings. In some situations, e.g., when a programming language does not provide built-in arrays or when a mapping has particular requirements, dynamic structures such as linked lists, trees or hash tables are suitable representations of a mapping.

Regardless of the underlying representation, a *mapping* is a total function $\mu : I \to V$ where I is a set of *indexes* and V is a set of *values*. The type of both the indexes and the values is arbitrary. A mapping is *injective* when distinct indexes are

∗ This research has been partially supported by the DAAD/NSF grant INT-9981317, the German Research Council (DFG) grant Ha 2457/1-2 and the NSF grants CCR-0110496 and CCR-0218224.

mapped to distinct values, if $i_1, i_2 \in I$ and $i_1 \neq i_2$, then $\mu(i_1) \neq \mu(i_2)$. A mapping is *finite* when the set of indexes is finite.

We make one of either two additional assumptions on the set V of values. The first assumption requires the *a priori* knowledge of a finite subset V' of V containing $\mu(I)$. If μ is a injective finite mapping, V' necessarily exists, since $\mu(I)$ has the same cardinality as I, and hence is finite. However, V' must be known *before* computing μ. This assumption trivially holds when V itself is finite. The second assumption, which is much weaker, requires the existence of an enumeration $v_0, v_1, \ldots$ of the values. Since in a program V is represented by either a primitive type or an algebraic type, the second assumption is easily satisfied for most problems.

In this paper we describe the design and implementation of an injective finite mapping suitable for a programming language with the following characteristics. The language is declarative, thus neither state updates nor side effects are allowed. The language is concurrent, thus different portions of the mapping can be computed concurrently and asynchronously by different portions of a program. This second characteristic has some non-trivial consequences that will be discussed later.

In a nutshell, our design represents the mapping as a collection of index-value pairs which can be structured as a list or a tree. It is expected that the program computes a function, e.g., for each index of a problem, the program computes a single value. The novelty is that in the *program representation* of the mapping the roles of an index and a value are swapped. This ensures that the inverse mapping is a function as well, hence the mapping is injective. Our design represents the mapping as an incomplete structure. This structure contains value-index pairs $\langle v, i \rangle$ (the problem maps i to v) where initially i is an uninstantiated variable. When a value-index pair of the representation, say $\langle v, i' \rangle$, is computed, the program attempts to unify i' with i. This ensures the injectivity condition, since the unification prevents two distinct indexes of the problem, i and i', to be mapped to the same value v. This also supports concurrency, since the unification is unaffected by the order in which value-index pairs are computed.

A class of puzzles known as cryptarithms is an ideal problem to discuss our design and implementation of an injective finite mapping: the mapping itself is the solution of the problem and it is convenient to compute index-value pairs of this mapping concurrently. The opportunity to compute the mapping concurrently be explained and motivated in Section 3.

The Merriam-Webster OnLine dictionary (Merriam-Webster, n.d.) defines a *cryptarithm* as "an arithmetic problem in which letters have been substituted for numbers and which is solved by finding all possible pairings of digits with letters that produce a numerically correct answer." A well-known example of cryptarithm is:

$$\texttt{SEND + MORE = MONEY} \tag{1}$$

Customarily, in a cryptarithm distinct letters stand for distinct digits and leading zeros are not allowed.

The solution of a cryptarithm is an injective finite mapping where the indexes are the letters occurring in the cryptarithm and the values are the digits. The solution of (1), graphically represented as a mapping, is shown below:

$$\begin{array}{ccccccccccccccc} \mathtt{S} & \mathtt{E} & \mathtt{N} & \mathtt{D} & + & \mathtt{M} & \mathtt{O} & \mathtt{R} & \mathtt{E} & = & \mathtt{M} & \mathtt{O} & \mathtt{N} & \mathtt{E} & \mathtt{Y} \\ \downarrow & \downarrow & \downarrow & \downarrow & & \downarrow & \downarrow & \downarrow & \downarrow & & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ \mathtt{9} & \mathtt{5} & \mathtt{6} & \mathtt{7} & + & \mathtt{1} & \mathtt{0} & \mathtt{8} & \mathtt{5} & = & \mathtt{1} & \mathtt{0} & \mathtt{6} & \mathtt{5} & \mathtt{2} \end{array}$$

A cryptarithm such as (1) in which letters form meaningful words, often in meaningful phrases, is referred to as an *alphametic*. There exist a large number of witty alphametics. Alphametics with a unique solution, such as (1), are more elegant, but a unique solution is not required. The following alphametic has 130 solutions:

```
TOO + MUCH = BEER
```

Solving a cryptarithm by brute force, e.g., by generating and testing every plausible mapping, is inefficient. Finite domain constraint solvers find solutions efficiently. Our program for cryptarithms is not as efficient, although it finds solutions in milliseconds. A brute force program using the same compiler/interpreter is over 150 times slower. We use cryptarithms as running examples. Of course, injective finite mappings are not confined to these puzzles and we hint at other applications as we go along.

This paper is structured as follows. Section 2 briefly recalls some principles of functional logic programming and the programming language Curry which we use to present the examples. Section 3 presents the design of an injective finite mapping in a functional logic program and its implementation in Curry. Section 5 concludes the paper. The complete programs are available on-line.

2 Functional Logic Programming and Curry

This section introduces both the basic ideas of functional logic programming and the elements of the programming language Curry (Hanus (ed.), 2003) that are necessary to understand the subsequent examples.

Functional logic programming integrates in a single programming model the most important features of functional and logic programming (see (Hanus, 1994) for a detailed survey). Functional logic languages feature algebraic datatypes, pattern matching, and logical variables. Supporting the latter requires some built-in search principle to guess the appropriate instantiations of logical variables. There exist many languages that are functional logic in this broad sense, e.g., Curry (Hanus (ed.), 2003), Escher (Lloyd, 1999), Le Fun (Aït-Kaci *et al.*, 1987), Life (Aït-Kaci, 1990), Mercury (Somogyi *et al.*, 1996), NUE-Prolog (Naish, 1991), Oz (Smolka, 1995), Toy (López-Fraguas & Sánchez-Hernández, 1999), among others.

One of the characteristic features of functional logic programming is the evaluation—particularly the *lazy* evaluation—of expressions containing logical variables. Both *narrowing* and *residuation* serve this purpose.

When an expression e cannot be evaluated due to the presence of an uninstantiated logical variable x, narrowing non-deterministically instantiates x to keep the evaluation of e from halting. By contrast, residuation suspends the evaluation of e, transfers control to another portion of the program, and resumes the evaluation of e if and when x becomes sufficiently instantiated.

Residuation is conceptually simple and relatively efficient, but incomplete. It is not always able to obtain the result of a computation. By contrast, narrowing is complete if an appropriate strategy (Antoy, 1997; Antoy *et al.*, 2000) is chosen, but it is potentially less efficient than residuation because of its propensity to generate a larger search space. Functional logic languages can be effective with either mechanism. Curry offers both residuation and narrowing in a unified computation model (Hanus, 1997). Each function is declared either *flexible* or *rigid.* Flexible functions narrow, whereas rigid functions residuate. Flexibility, through narrowing, enables a function both to "run backward." The arguments of a function can be computed from the result to solve arbitrary equational constraints over user-defined datatypes.

Curry has a Haskell-like (Peyton Jones & Hughes, 1999) syntax, e.g., (type) variables and function names usually start with lowercase letters and the names of type and data constructors start with an uppercase letter. The application of f to e is denoted by juxtaposition ("$f\ e$"). In addition to Haskell, Curry supports logic programming by means of *free* (logical) variables in both conditions and right-hand sides of defining rules. Thus, a Curry *program* consists of the definition of functions and the declaration of data types on which the functions operate. Functions are evaluated lazily and can be called with partially instantiated arguments. In general, functions are defined by conditional equations, or *rules*, of the form:

$$f\ t_1 \ldots t_n\ \texttt{|}\ c\ \texttt{=}\ e\quad \texttt{where}\ vs\ \texttt{free}$$

where $t_1, \ldots, t_n$ are *data terms* (i.e., terms without defined function symbols), the *condition* c is either a Boolean expression or a constraint, e is an expression and the `where` clause introduces a set of *free* variables. These variables are used as uninstantiated (unknown) arguments of function applications in the condition and/or the right-hand side of the rule. We will show shortly that this programming style reduces the amount of details necessary to encode a problem into a program.

Both the condition and the `where` clause of a rule are optional. Curry predefines *equational constraints* of the form $e_1 \texttt{=:=} e_2$ which is satisfiable if both sides e_1 and e_2 are evaluated to unifiable data terms. Furthermore, "c_1 `&` c_2" denotes the *concurrent conjunction* of the constraints c_1 and c_2. Concurrent means that the evaluation steps of each conjunct can be interleaved. Interleaving the steps may become essential to ensure the completeness of the computation, if one conjunct residuates.

The following example, although contrived, shows the behavior of residuation and narrowing, and their interplay through concurrency. Consider the conditional rule:

```
goal x y | x+x =:= y & x =:= 2  = y
```

Uninstantiated variables in goals are denoted by upper-case identifiers. The expression `goal X Y` is evaluated as follows. The execution of the first constraint demands the evaluation of `X+X`. Since the operation "+" is rigid and `X` is unbound, the evaluation residuates and the control goes to the second constraint. Since the operation "`=:=`" is flexible, `X` is bound to `2`. The evaluation of `X+X` resumes and yields `4`. Thus, `Y` is bound to `4` and returned.

Similar to Haskell, the `where` clause can also contain local definitions. In contrast to Haskell, where the first matching function rule is applied, in Curry all matching (to be more precise, unifiable) rules are non-deterministically applied to support logic programming. This enables the definition of non-deterministic functions which may have more than one result for a given input. As an example consider the function:

```
insert :: a -> [a] -> [a]
insert e []     = [e]
insert e (x:xs) = e : x : xs
insert e (x:xs) = x : insert e xs
```

which inserts an element into a list at some non-deterministically chosen position.

The second and third rules defining `insert` overlap. As a consequence, the expression `(insert 1 [3,5])` has three values: `[1,3,5]`, `[3,1,5]`, and `[3,5,1]`. Using `insert`, we can define a permutation of a list as follows:

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

As an example of solving constraints, we define a function that checks whether a word embeds the letters of another word, e.g, "care" embeds "ace", and returns the unused letters, "r" in this example, in some order. For this purpose we use the concatenation of two lists which is denoted by the infix operator "`++`". The operator "`++`" is *flexible*. The application of "`++`" may instantiate variables in the arguments, if this is necessary to execute a computation step. By default, functions are flexible with a few exceptions, e.g., I/O actions and arithmetic operations on numbers (see (Hanus (ed.), 2003) for details).

Now we define the required function by a single conditional rule:

```
embeds w1 w2 | perm w1 =:= w2 ++ w3
             = w3
             where w3 free
```

As apparent from the function `insert`, Curry allows coding functions, called non-deterministic, that return more than one value for the same arguments. E.g., the infix operator `!` non-deterministically returns one of its arguments. It is defined by the two rules:

```
x ! y = x
x ! y = y
```

The operational semantics of Curry, precisely described in (Hanus, 1997; Hanus (ed.), 2003), is a conservative extension of both lazy functional programming (if no free variables occur in the program or the initial goal) and (concurrent) logic programming. Since computations are based on an optimal evaluation strategy (Antoy, 1997;

Antoy *et al.*, 2000), Curry can be considered a generalization of concurrent constraint programming (Saraswat, 1993) with a lazy (optimal) evaluation strategy. Furthermore, Curry also offers features for application programming like modules, monadic I/O, ports for distributed programming, and specialized libraries.

There exist several implementations of Curry. The examples presented in this paper were all compiled and executed by PAKCS (Hanus *et al.*, 2003), a compiler/interpreter for a large subset of Curry.

3 Design and Implementation of the Mapping

A plausible implementation of a finite mapping is any structure defining index-value pairs, e.g., an array, a list of pairs, etc. Index-value pairs are computed during the execution of a program. Injectivity means that if two pairs have the same value they also have the same index. Thus, the algorithm that ensures injectivity must compare any computed index-value pair with every other pair previously computed. If a pair with the same value was previously computed, the indexes in the previous and the new pairs must be equal.

A problem with this algorithm arises if a functional logic program computes index-value pairs concurrently, e.g., due to concurrent constraint solving. This condition prevents one from sequentially passing a partially constructed mapping through the portions of a program computing index-value pairs to ensure that a newly computed pair does not violate the injectivity condition. In what follows, we show a technique for solving this problem. We make our technique more concrete by discussing the architecture of a simple program to solve a cryptarithm.

A program to solve (1) declares one variable for each letter. Initially, these variables are uninstantiated:

```
s,e,n,d,m,o,r,y free
```

The solution of the problem is a suitable instantiation of these variables, which implicitly defines the mapping which is the subject of this paper. The instantiation of each variable is determined by Equation (1). This equation can be processed as a single unit or it can be broken into a set of "smaller" equations. These smaller equations establish the conditions that the letters must satisfy for the column of the units, the tens, the hundreds, etc., exactly as one would perform the addition by hand. The following display depicts the situation:

```
c3 c2 c1 c0
   s  e  n  d +
   m  o  r  e =
  -------------
 m  o  n  e  y
```
(2)

where ci, for $i = 0, 1, 2, 3$, is a carry. These equations are coded as:

```
      c3 =:= m
s+m+c2 =:= c3*10+o
```

```
e+o+c1 =:= c2*10+n
n+r+c0 =:= c1*10+e
d+e    =:= c0*10+y
```

Each carry must be either 0 or 1 and consequently it is (non-deterministically) initialized as follows:

ci = 0!1 $\qquad\qquad i = 0, \ldots, 3$

It follows from the conventions of the problem that `m` is not zero and consequently `c3` is equal to one. Our simple program ignores this precise inference. However, this equation together with the equation defining the carry constrain the possible values of `m` to zero and one only.

Splitting the problem's equation into a set of smaller equations is a slight complication, since it requires the introduction of additional variables for the carries. However, a set of smaller equations has a significant advantage. With appropriate control, the program detects the instantiation of a variable that does not satisfy some equation, when fewer variables are instantiated. This considerably speeds up the execution of the program but it introduces a substantial complication.

The solutions of the five equations are computed concurrently. The order in which the solution of each equation is computed is undetermined. Since the variables, `s`, `e`, ... are initially unbound and the addition and multiplication operators residuate, the execution of the equations that the variables must satisfy is suspended until both the operands of an operator become bound. Each variable is non-deterministically bound to a digit, similarly to the carries. In this case, though, the choice ranges over every digit, or every positive digit for `m` and `s`. It is inappropriate to non-deterministically instantiate the variables as it is done for the carries, e.g.:

```
d =:= 0!1!2!3!4!5!6!7!8!9
...
m =:= 1!2!3!4!5!6!7!8!9
```

since this does not ensure that distinct variables are bound to distinct digits. It is also inappropriate to pass around a structure containing the current binding of the variables, since the order in which the variables will be instantiated cannot be easily determined in advance. Here is where our ideas make a difference.

We represent the mapping as a list referred to as the *store*. The store is indexed by the *values* of the problem. For cryptarithms, this indexing is natural and straightforward since the values are the digits `0,1,...,9`. Initially, the elements of the store are free variables. The elements in the store are referred to as *tokens*. Putting a token into the store represents the action of choosing a value that must be different from the value of any other choice. The type of the tokens is arbitrary. Often, it is convenient to represent the tokens with the *indexes* of the problem. For the alphametic (1), we choose the characters `S`, `E`, `N`, ... as the tokens.

Thus, the indexes and values of a problem are used as values and indexes, respectively, in the store. The roles they have in the problem is swapped in the store. We will shortly explain why this reversal of roles is a natural and necessary aspect of the design. In the particular case of the alphametic (1), the set of values is finite. This enables us to create the store when the execution of the program begins. At the end of the only successful computation for our example (note that in general there can be more than one successful computation), the content of the store is shown below, where "•" represents an uninstantiated variable:

O	M	Y	•	•	E	N	D	R	S

Thus, in the program, the initial store is a list of 10 free variables:

```
store = [s0,s1,s2,s3,s4,s5,s6,s7,s8,s9]
      where s0,s1,s2,s3,s4,s5,s6,s7,s8,s9 free
```
(3)

Elements of the list can be accessed by the list index operator, `!!`, defined in the *prelude* as:

```
(!!) :: [a] -> Int -> a
(x:xs) !! n | n == 0 = x
            | n >  0 = xs !! (n-1)
```

A letter of the cryptarithm is paired to a digit by the function `digit` defined as follows:

```
digit token | store !! x =:= token
            = x
            where x = 0!1!2!3!4!5!6!7!8!9
```

Although the associated digit is non-deterministically selected, the condition on the store ensures the injectivity of the mapping. The argument `token` must be unique for each letter, hence, it is natural and convenient to represent it with the letter itself—a character in the program.

Thus, the letters of the cryptarithm are nondeterministically instantiated as follows:

```
s =:= nzdigit 'S'
e =:= digit   'E'
n =:= digit   'N'
...
```

where `nzdigit` is a variant of `digit` that returns only non-zero digits. For example, `digit 'Y'` returns 2 if and only if the second (counting from zero) element of the store is bound to `'Y'`. The entire program for this problem is shown in the appendix. Programs with a different representation of the store are available on-line at URL `http://www.cs.pdx.edu/~antoy/flp/patterns/distinct-choices-dir/`.

The reversal of the roles of indexes and values in the store may be confusing at first, but it has a natural explanation. The injectivity requirement of a mapping μ is intended to *prevent* the condition in which two distinct indexes, say l and m, satisfy $\mu(l) = v = \mu(m)$, for some value v. In the store, the value v is associated to some value, a digit i, of the problem. Specifically, the variable v is the i-th element of the store. Every time an index of the problem, some letter L, is mapped to i, the program attempts to unify, hence instantiate, the variable v to L. The attempt succeeds if and only if either v was uninstantiated, or v was instantiated to L already. Thus, no two distinct indexes of the problem can be mapped to the same value of the problem.

4 Variations

In the program that we are discussing, the association between a variable v of the store and a value i of the problem is positional. The store is a list and the variable v is the i-th element of the list. The store is constructed by (3), since the set of values of the problem is finite and known in advance. There are many variations of this design. We discuss two of these variations below. The first variation constructs the store lazily. This is useful when no finite set of values is known in advance. The second variation uses a tree-like structure for representing the mapping. This may provide a better access performance for some problems.

The first variation requires a small change to the code. All it takes to construct the store lazily is to replace (3) with the following:

```
store free
```
(4)

The list index operator shown earlier is *flexible*. The evaluation of an expression of the form $l \,!!\, n$ instantiates enough l, if necessary, to ensure the access to its n-th element. In other words, the elements in the store are produced on demand rather than up-front.

This variation is interesting when the set of values of the problem is infinite. The store is indexed by the values of the problem, which in general will not be natural numbers. In this case, we assume the existence of an enumeration $v_0, v_1, \ldots$ of the values. The enumeration implicitly defines an indexing function: a value v_i of the problem indexes the i-th element of the store. For a cryptarithm, this indexing is trivial since the values of the problem are the digits, which can directly index the store.

The mapping can be implemented as Curry module:

```
module ListFM (mapto) where
mapto :: [a] -> Int -> a -> Success
mapto fm i v = fm !! i =:= v
```

A functional logic language such as Curry computes with narrowing. Therefore, the operation `mapto` can be used to extend the store with a new index-value pair, to verify that an index-value pair is in the store, and to retrieve the value of an index-value pair from an index.

The second design variation represents the store as a binary trie. A value is addressed using the binary representation of the index, least significant bit first, to select a branch of the trie.

```
module TreeFM (mapto) where
data Tree a = Tree a (Tree a) (Tree a)
mapto :: Tree a -> Int -> a -> Success
mapto (Tree w l r) i v
  = if i == 0                                           (5)
    then w =:= v
    else mapto s j v
      where j = i `div` 2
            s = if i `mod` 2 == 1 then l else r
```

Since the program defines the store as a free variable, declaration (4) is independent of the representation of the store. In (5), the representation of the store is hidden to the program.

Different representations of the store have different computational characteristics. For example, a trie representation may be more efficient than a list for some problem, since for balanced tries the average access time of an element is $O(log\ n)$ instead of $O(n)$. The size of a cryptarithm is, of course, too small to justify this representation. Informal benchmarks show no significant differences in execution time when the store is represented by a list or by a trie.

The crucial feature of the injective finite mapping that we are discussing is the possibility of concurrently computing index-value pairs. However, the proposed design can be employed also in problems where concurrency is not an issue. Some examples are available on-line at URL `http://www.cs.pdx.edu/~antoy/flp/patterns/distinct-choices-dir/`.

As we already mentioned, splitting a problem into smaller parts that are solved concurrently has the advantage that wrong choices, expressed as instantiations of some variables that cannot lead to a solution, are detected earlier. This can lead to a considerable reduction of the search space. For instance, a naive functional solution to our cryptarithm (enumerating all the digits and testing Equation (1)) has an unacceptable execution time. This solution can be improved by merging partial tests with the enumeration of values in a sophisticated way. However, the resulting code is less concise and more difficult to generalize than our concurrent implementation of injective finite mappings. A similar argument holds for a naive logic programming solution. Although Prolog does not offer concurrency, various implementations of Prolog support concurrent extensions. These extensions enable a similar implementation of our ideas, but with less concise code due to the missing functional notation. Constraint logic programming with finite domains provide more efficient computations thanks to efficient built-in constraint solvers. However, our approach can be easily generalized to infinite domains, as discussed above.

5 Conclusion

Functional logic programs, in addition to ordinary functional computations, provide both concurrency and logic variables. Concurrency supports a powerful and expressive programming style, but it complicates some tasks, in particular the computation of an injective finite mapping. We have presented the design and implementation of one such mapping for a functional logic language.

The design relies on a representation of index-value pairs where the values of the problem play the role of indexes in the representation and the indexes of the problem are initially unbound variables. During the computation, the variables of the representation are non-deterministically bound to the indexes of the problem. This design ensures the injectivity of the mapping even when index-value pairs are computed concurrently and independently.

Acknowledgment

We are grateful to the reviewers for many suggestions that improved the final version.

References

Aït-Kaci, H. (1990) An overview of LIFE. Schmidt, J. and Stogny, A. (eds), *Proc. Workshop on Next Generation Information System Technology* pp. 42–58. Springer LNCS 504.

Aït-Kaci, H., Lincoln, P. and Nasr, R. (1987) Le Fun: Logic, equations, and functions. *Proc. 4th IEEE Internat. Symposium on Logic Programming* pp. 17–23.

Antoy, S. (1997) Optimal non-deterministic functional logic computations. *Proc. International Conference on Algebraic and Logic Programming (ALP'97)* pp. 16–30. Springer LNCS 1298.

Antoy, S., Echahed, R. and Hanus, M. (2000) A needed narrowing strategy. *Journal of the ACM* **47**(4):776–822.

Hanus, M. (1994) The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming* **19&20**:583–628.

Hanus, M. (1997) A unified computation model for functional and logic programming. *Proc. of the 24th ACM Symposium on Principles of Programming Languages (Paris)* pp. 80–93.

Hanus, M., Antoy, S., Engelke, M., Höppner, K., Koj, J., Niederau, P., Sadre, R. and Steiner, F. (2003) *PAKCS: The Portland Aachen Kiel Curry System.* Available at `http://www.informatik.uni-kiel.de/~pakcs`.

Hanus (ed.), M. (2003) *Curry: An Integrated Functional Logic Language (Vers. 0.8).* Available at `http://www.informatik.uni-kiel.de/~curry`.

Lloyd, J. (1999) Programming in an integrated functional and logic language. *Journal of Functional and Logic Programming* 1–49.

López-Fraguas, F. and Sánchez-Hernández, J. (1999) TOY: A Multiparadigm Declarative System. *Proc. of RTA'99* pp. 244–247. Springer LNCS 1631.

Merriam-Webster. *Dictionary OnLine.* Web document. URL `http://www.m-w.com/dictionary.htm` accessed August 28, 2003 05:56 UTC.

Naish, L. (1991) Adding equations to NU-Prolog. *Proc. of the 3rd Int. Symposium on Programming Language Implementation and Logic Programming* pp. 15–26. Springer LNCS 528.

Peyton Jones, S. and Hughes, J. (1999) *Haskell 98: A Non-strict, Purely Functional Language.* `http://www.haskell.org`.

Saraswat, V. (1993) *Concurrent Constraint Programming.* MIT Press.

Smolka, G. (1995) The Oz programming model. van Leeuwen, J. (ed), *Computer Science Today: Recent Trends and Developments* pp. 324–343. Springer LNCS 1000.

Somogyi, Z., Henderson, F. and Conway, T. (1996) The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming* **29**(1-3):17–64.

Appendix

Complete implementation of the SEND+MORE=MONEY cryptarithm.

```
infixl 0 !
x ! _ = x
_ ! y = y

solve | equations ()
  = "\n" ++
    " "    ++ show s ++ show e ++ show n ++ show d ++ "\n" ++
    " "    ++ show m ++ show o ++ show r ++ show e ++ "\n" ++
    show m ++ show o ++ show n ++ show e ++ show y ++ "\n"
  where
      store = [s0,s1,s2,s3,s4,s5,s6,s7,s8,s9]
         where s0,s1,s2,s3,s4,s5,s6,s7,s8,s9 free

      -- the digits
      s,e,n,d,m,o,r,y free

      -- the carries
      c0 = 0!1
      c1 = 0!1
      c2 = 0!1
      c3 = 0!1

      -- the problem's relations, fragmentation is good
      -- unused argument avoids circular patterns test
      equations _ = c3 =:= m &
              s+m+c2 =:= c3*10+o &
              e+o+c1 =:= c2*10+n &
              n+r+c0 =:= c1*10+e &
              d+e    =:= c0*10+y &
              m =:= nzdigit 'M' &
              s =:= nzdigit 'S' &
              o =:= digit   'O' &
              e =:= digit   'E' &
              n =:= digit   'N' &
              r =:= digit   'R' &
              d =:= digit   'D' &
              y =:= digit   'Y'

      nzdigit token | store !! x =:= token = x
          where x = 1!2!3!4!5!6!7!8!9
      digit token | store !! x  =:= token = x
          where x = 0!1!2!3!4!5!6!7!8!9
```