©Springer-Verlag
In *Proc. of the 6th International Symposium on Functional and Logic Programming (FLOPS 2002)*, Aizu (Japan). Springer LNCS 2441, pp. 67–87, 2002

Functional Logic Design Patterns[⋆]

Sergio Antoy[1] Michael Hanus[2]

[1] Computer Science Department, Portland State University,
P.O. Box 751, Portland, OR 97207, U.S.A.
`antoy@cs.pdx.edu`

[2] Institut für Informatik, Christian-Albrechts-Universität Kiel
Olshausenstr. 40, D-24098 Kiel, Germany
`mh@informatik.uni-kiel.de`

Abstract. We introduce a handful of software design patterns for functional logic languages. Following usual approaches, for each pattern we propose a name and we describe its intent, applicability, structure, consequences, etc. Our patterns deal with data type construction, identifier declarations, mappings, search, non-determinism and other fundamental aspects of the design and implementation of programs. We present some problems and we show fragments of programs that solve these problems using our patterns. The programming language of our examples is Curry. The complete programs are available on-line.

1 Introduction

A *design pattern* is a proven solution to a recurring problem in software design and development. A pattern itself is not primarily code. Rather it is an expression of design decisions affecting the architecture of a software system. A pattern consists of both ideas and recipes for the implementations of these ideas often in a particular language or paradigm. The ideas are reusable, whereas their implementations may have to be customized for each problem. For example, a pattern introduced in this paper expresses the idea of calling a data constructor exclusively indirectly through an intermediate function. The idea is applicable to a variety of problems, but the code of the intermediate function is highly dependent on each application.

Patterns originated from the development of object-oriented software [7] and became both a popular practice and an engineering discipline after [14]. As the landscape of programming languages evolves, patterns are "translated" from one language into another [12,16]. Some patterns are primarily language specific, whereas others are fundamental enough to be largely independent of the language or programming paradigm in which they are coded. For example, the *Adapter* pattern, which solves the problem of adapting a service to a client coded for different interface, is language independent. The *Facade* pattern, which presents a set of separately coded services as a single unit, depends more on the modularization features of a language than the language's paradigm itself. The *Composite* and *Visitor* patterns are critically dependent on features of object

⋆ 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 grant CCR-0110496.

orientation, such as derivation and overriding, in the sense that other paradigms tend to trivialize them. For example, Section 4 references programs that show the simplicity of these patterns in a declarative paradigm.

In this paper, we present a handful of patterns intended for a declarative paradigm—in most cases specifically for a functional logic one. High level languages are better suited for the implementation of reusable code than imperative languages, see, e.g., parser combinators [10]. Although in some cases, e.g., search-related patterns, we attempt to provide reusable code, the focus of our presentation is on the reusability of design and architecture which are more general than the code itself. Our presentation of a pattern follows the usual (metapattern) approaches that provides, e.g., name, intent, applicability, structure, consequences, etc. Some typical elements, such as "known uses," are sparse or missing because of the novelty of our work. To our knowledge this is the first paper on patterns for functional logic programming.

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 a small catalog of functional logic patterns together with motivating problems and implementation fragments. Section 4 contains references to on-line versions of the entire programs whose fragments are interspersed in the text. Section 5 concludes the paper. Appendix A further elaborates a framework for search problems analyzed in relation to the Incremental Solution pattern.

2 Functional Logic Programming and Curry

This section provides both an introduction to the basic ideas of functional logic programming and the elements of the programming language Curry 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 [18] for a detailed survey). Thus, functional logic languages provide pattern matching in the definition of functions and predicates as well as the use of logical variables in expressions. The latter feature requires some built-in search principle in order to guess the appropriate instantiations of logical variables. There are a number of languages supporting functional logic programming in this broad sense, e.g., Curry [28], Escher [31], Le Fun [2], Life [1], Mercury [39], NUE-Prolog [34], Oz [38], Toy [32], among others. Some of these proposals, as well as some Prolog implementations (e.g., Ciao Prolog [9]), consider the functional notation only as syntactic sugar for particular predicates. Functions with n arguments are translated into predicates of arity $n+1$ by including the result value as an additional argument. This technique, known as naive *flattening*, does not exploit the fact that in some situations the value of an argument of a function does not affect the result. Computing these arguments would be a waste. For instance, the *demand-driven* or *needed* evaluation strategy evaluates an argument of a function only if its value is needed to compute the result. An appropriate notion of "need" is a subtle point in the presence of a non-deterministic choice, e.g., a typical computation step, since the need of an argument might depend on the choice itself. [4] proposes a *needed narrowing* strategy to solve this problem. This strategy is optimal w.r.t. both the length of successful derivations and the disjointness of computed solutions. Narrowing is a combination of term reduction as in functional programming and (non-deterministic) variable instantiation as in logic programming. It exploits the presence of functions without transforming them into predicates which yields a more efficient operational behavior (e.g., see [17,20]).

Functions can be also interpreted as a declarative notion to improve control in logic computations, i.e., functional logic languages do not include non-declarative control primitives like the Prolog's "cut" operator. The main reason is the dependency between input and output arguments which leads to needed evaluations. It is interesting to note that the *determinism* property of functions (i.e., there is at most one result value for fixed input arguments) is not strictly required. Actually, one can also deal with *non-deterministic functions* that deliver more than one result on a given input [3,15]. This comes at no cost to the implementor since non-determinism is always available in functional logic languages. On the other hand, the combination of demand-driven evaluation and non-deterministic functions can result in a large reduction of the search space [3,15].

It is well known that narrowing is an evaluation strategy that enjoys soundness and completeness in the sense of functional and logic programming, i.e., computed solutions/values are correct and correct solutions/values are computed. Nevertheless, narrowing is not able to deal with external functions. Therefore, [1,2,8,31,34] proposed alternative evaluation strategies based on *residuation*. The residuation principle delays a function call until the arguments are sufficiently instantiated so that the call can be deterministically reduced. Thus, residuation-based languages also support concurrent evaluations in order to deal with suspended computations. However, they do not ensure completeness in contrast to narrowing [19].

[21] proposes a seamless combination of needed narrowing with residuation-based concurrency. This is the basis of the programming language Curry [28], an attempt to provide a standard in the area of functional logic programming languages. The patterns presented in this paper are independent of Curry since they require only general features that can be found in many functional logic languages. Nevertheless, we use Curry for its dominant role in this field and due to its support for all typical features of functional logic languages. Thus, we provide in the following a short overview on Curry.

Curry has a Haskell-like syntax [35], i.e., (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 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 a Boolean function or constraint, e is an expression and the `where` clause introduces a set of free variables. The condition and the `where` clause can be omitted. Curry defines *equational constraints* of the form e_1 `=:=` e_2 which are satisfiable if both sides e_1 and e_2 are reducible to unifiable data terms. Furthermore, "c_1 `&` c_2" denotes the *concurrent conjunction* of the constraints c_1 and c_2 which is evaluated by solving both c_1 and c_2 concurrently.

The `where` clause introduces the free variables vs occurring in c and/or e but not in the left-hand side. Similarly to Haskell, the `where` clause can also contain other local function or pattern 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 on a given input.

Consider the following definition of lists and a function that non-deterministically inserts an element into a list:

```
data List a = [] | a : List a

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

The data type declaration defines `[]` (empty list) and `:` (non-empty list) as the constructors of polymorphic lists. The symbol `a` is a type variable ranging over all types and the type "`List a`" is usually written as `[a]` for conformity with Haskell. The second line of the code declares the type of the function `insert`. This declaration is optional, since the compiler can infer it, and its is written for checkable redundancy. The type expression α`->`β denotes the type of all functions from type α to type β. Since the application of a function is curried, `insert` takes an element of type `a`, a list of elements of type `a` and returns a list of elements of type `a`, where `a` is any type.

A consequence of the rules defining `insert`, where the second and third rule overlap, is that the expression `(insert 1 [3,5])` has three possible values: `[1,3,5]`, `[3,1,5]`, and `[3,5,1]`. Using `insert`, we define the possible permutations of a list by:

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

As an example of solving constraints, we want to define a function that checks whether some list starts with a permutation of another list and delivers the list of the remaining elements. For this purpose we use the concatenation of two lists which is defined by:

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

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

```
pprefix xs ys | conc (perm ys) zs =:= xs
              = zs                          where zs free
```

The operational semantics of Curry, precisely described in [21,28], 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 it is based on an optimal evaluation strategy [4], Curry can be considered a generalization of concurrent constraint programming [36] with a lazy (optimal) evaluation strategy. Furthermore, Curry also offers features for application programming like modules, monadic I/O, encapsulated search [27], ports for distributed programming [22], libraries for GUI [23] and HTML programming [24] etc. We do not discuss them here since they are not relevant for the subsequent examples.

There exist several implementations of Curry. The examples presented in this paper were all compiled and executed by PAKCS [25], a compiler/interpreter for a large subset of Curry.

3 Patterns

In this section we present a small catalog of patterns. Practitioners makes a distinction between the words *pattern* and *idiom*, although there are no formal definitions. An idiom is more language specific and addresses a smaller and less general problem than a pattern. An emblematic idiom of the C programming language is the code for copying a string:

```
while(*s++ = *t++) ;
```

The problem solved by this code is simple and the code relies on several peculiarities of the C language, e.g., the combination of the conventions of ending strings with a null character and representing the Boolean value *false* with the integer value zero. By contrast, patterns are more general both in applicability and scope. Many patterns of [14] were originally coded in C++ and/or Smalltalk. Later, they were easily ported to other object-oriented languages such as Java [16,30]. Also, these patterns address design and/or architectural problems that often span across several classes with several methods each. The code of these classes and methods often depends on a specific application and, consequently, it is not easily reusable.

The same distinction between patterns and idioms holds for functional logic languages. To clarify the difference, we discuss an idiom in Curry. This idiom ensures that a function returns a value only if the value satisfies a certain property. The idiom is based on the following `suchthat` operator:

```
infix 0 `suchthat`
suchthat :: a -> (a->Bool) -> a
x `suchthat` p | p x = x
```

For example, typical functional logic implementations of the n-queens puzzle generate a permutation of the input and test whether such permutation is “safe.” Using the `suchthat` operator, the top-level function, `queens`, of an implementation of this puzzle looks like:

```
queens x = permute x `suchthat` safe
```

where `permute` and `safe` have the expected meanings.

When appropriately used, this idiom yields terser and more elegant code, but hardly more than this. The patterns that we describe below address non-trivial solutions of some general and challenging problems.

3.1 Constrained Constructor

Name	*Constrained Constructor*
Intent	prevent invoking a constructor that might create invalid data
Applicability	a type is too general for a problem
Structure	define a function that either invokes a constructor or fails
Consequences	invalid instances of a type are never created by the function
Known uses	
See also	[30]; sometimes used with the Incremental Solution

The signature of a functional logic program is partitioned into *defined operations* and *data constructors*. They differ in that operations manipulate data by means of rewrite rules, whereas constructors create data and have no associated rewrite rules. Therefore, a constructor symbol cannot perform any checks on the arguments to which it is applied. If a constructor is invoked with arguments of the correct types, but inappropriate values, conceptually invalid data is created. We use an example to clarify this point.

The *Missionaries and Cannibals* puzzle is stated as follows. Three missionaries and three cannibals want to cross a river with a boat that holds up to two people. Furthermore, the missionaries, if any, on either bank of the river cannot be outnumbered by the cannibals (otherwise, as the intuition hints, they would be eaten by the cannibals).

A state of this puzzle is represented by the number of missionaries and cannibals and the presence of the boat on an arbitrarily chosen bank of the river, by convention the *initial* one:

```
data State = State Int Int Bool
```

For example, with suitable conventions, `(State 3 3 True)` represents the initial state. The simplicity of this representation has the drawback that invalid states, e.g., those with more than 6 people, can be created as well. Unless complex and possibly inefficient types for the state are defined, it is not possible to avoid the creation of invalid states using constructors alone.

Before completing the presentation of the *Constrained Constructor* pattern, consider one of the rewrite rules defining the operation that moves the boat and some people across the river:

```
move (State m c True)
  | m>=2 && (m-2==0 || m-2>=c) && (c==3 || m-2=<c)
  = State (m-2) c False           -- move 2 missionaries
...
```

This rewrite rule abstracts moving two missionaries across the river. The complex guard ensures that before the move there are at least two missionaries on the originating bank of the river, `m>=2`, and that after the move, on each bank, the missionaries are not outnumbered by the cannibals. The second conjunct of the condition ensures that after the move either there are no missionaries left on the originating bank of the river, `m-2==0`, or the number of missionaries left is not smaller than the number of cannibals, `m-2>=c`. The third conjunct expresses a similar condition for the destination bank. Nine other rewrite rules with different, but similarly complex, guards are required to complete the definition of this operation.

Both the complexity of operation `move` and the creation of invalid states are avoided by the *Constrained Constructor* pattern. This pattern ensures that only states that are consistent with the conditions of the puzzle and are safe for the missionaries are created. For example, `(State 2 1 _)` is not safe since on one bank of the river the cannibals outnumber the missionaries and therefore should not be created. The function that constructs only desirable states is defined below:

```
makeState m c b | valid && safe = State m c b
   where valid = 0<=m && m<=3 && 0<=c && c<=3
         safe  = m==3 || m==0 || m==c
```

Using operation `makeState`, the definition of operation `move` is greatly simplified with a minimal loss of efficiency:[1]

```
move (State m c True)
   = makeState (m-2) c False      -- move 2 missionaries
   ! makeState (m-1) c False      -- move 1 missionary
   ! makeState m (c-2) False      -- move 2 cannibals
   ! ...
```

Program *mission.curry*, referenced in Section 4, contains a second occurrence of the *Constrained Constructor* pattern. The program finds the solutions of the puzzle by constructing paths from the initial state (all the people and the boat on the initial bank) to the final state (no people and no boat on the initial bank). A path is defined by sequence of states as follows:

```
data Path = Initial State | Extend Path State
```

However, not all sequences of states are valid or desirable. In any path, any state except the initial one must be obtained from the preceding state by means of a move. Moreover, cycles in a path are undesirable since they unnecessarily consume memory and increase the size of the search space. Therefore, we have a second opportunity to use the *Constrained Constructor* pattern.

A crucial advantage of creating only valid paths with no cycles is that the search space of the puzzle changes from infinite to finite. This condition ensures that even naive strategies, e.g., depth-first search which may result from an incomplete implementation of a functional logic language such as a Prolog-based implementation [5], suffice to solve the puzzle. An implementation of the n-queens puzzle using the *Constrained Constructor* pattern, *queens.curry*, is referenced in Section 4. This implementation is

[1] The infix operator `!` denotes the most fundamental non-deterministic function. It returns one of its arguments and is defined by the two rules:

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

structurally identical to that of the missionaries and cannibals and is much faster than implementations, e.g., *queens-permute.curry*, of the n-queens puzzle based directly on permutations.

This pattern, in the form here presented, is not available in functional languages. In a functional program, if a function call fails the entire execution of the program fails. The same problem does not occur in a logic language, though a new problem arises. In a functional language, constructor and function symbols are syntactically interchangeable in an expression. In a logic language they are not. Thus, replacing a constructor with a constrained constructor changes the structure of a logic program.

3.2 Concurrent Distinct Choices

Name	*Concurrent Distinct Choices*
Intent	ensure that a mapping from indexes to values is injective
Applicability	index-value pairs are computed concurrently
Structure	bind a unique token to a variable indexed by a value
Consequences	the index-value relation is an injective mapping
Known uses	
See also	

A *injective mapping* is a function from a set of *indexes* to a set of *values* such that distinct indexes are mapped to distinct values. Defining one such mapping is a component of the solution of many problems. For example, programs for both the n-queens and cryptarithmetic puzzles that are based on injective mappings are referenced in Section 4. A plausible representation of an injective mapping consists of a structure containing index-value pairs. Index-value pairs are computed during the execution of a program. To ensure injectivity, when an index-value pair is computed the program must check that no previously computed pair with the same index has the same value.

The *Concurrent Distinct Choices* pattern serves this purpose. One noteworthy feature of this pattern is that it allows the concurrent computation of the mapping. In other words, different portions of a program can compute index-value pairs in a non-sequential flow of control (e.g., due to residuation). With this condition, an implementation cannot pass around the structure containing the current index-value pairs in order to test the injectivity constraint.

In the simplest form of this pattern, the values are integer numbers in the range 0 to $n-1$. The representation of the mapping is a list referred to as the *store*. Initially, the elements of the store are n free variables. The *values* are used as indexes in the store. The elements in the store are referred to as *tokens*. A token 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, the tokens are the *indexes* of the problem's mapping. Thus, the indexes and values of a problem are used as values and indexes respectively, i.e., the roles they have in the problem is reversed in the store.

To clarify this architecture, let us consider an example. A *cryptarithmetic puzzle* presents an arithmetic computation in which digits are replaced by letters. The problem is to find a correspondence from letters to digits that satisfies the computation. Different letters stand for different digits and leading zeros are disallowed in the encrypted representation of numbers. A well-know cryptarithmetic puzzle and its solution are shown in the following display:

```
SEND + MORE = MONEY
9567 + 1085 = 10652
```

In this case, the letters S, E, N, ... are mapped to the digits 9, 5, 6, ...

A program for this cryptarithmetic puzzle, *send-more.curry*, declares one variable for each letter:

```
vs,ve,vn,vd,vm,vo,vr,vy free
```

and defines, as a constraint, the set of equations that the variables must satisfy:

```
vd+ve     =:= c0*10+vy  &
vn+vr+c0 =:= c1*10+ve  &
ve+vo+c1 =:= c2*10+vn  &
vs+vm+c2 =:= c3*10+vo  &
      c3 =:= vm
```

where c0 is the carry of the units, c1 of the tens, etc. Each carry must be either 0 or 1 and consequently it is initialized as follows:

```
c_i = 0!1                    i = 0,...,3
```

It follows from the conventions of the problem that vm is not zero and consequently c3 is equal to one. In general, such precise inferences are not available and thus the program will ignore them.

The relation among the letters could be formulated as the single equation:

```
vd+ve + 10*(vn+vr) + 100*(ve+vo) + 1000*(vs+vm)
    =:= vy + 10*ve + 100*vn + 1000*vo + 10000*vm
```

Instead, we choose to split it into the conjunction of five simpler equations. This splitting enables the program to detect an incorrect mapping of letters to digits when fewer letters are mapped and consequently it improves considerably the efficiency of the execution. We will see shortly that these equations are not executed sequentially.

Since the variables, vs, ve, ..., that stand for the letters of the puzzle are initially unbound and the addition and multiplication operators are rigid, the execution of the equations that the variables must satisfy residuates, i.e., it is suspended until both the operands of an operator become bound. Each operand is a variable that is non-deterministically bound to a digit, similarly to the carries. However, in this case, different variables must be bound to different digits and the order in which variables are bound is not easily determined in advance. Here is where the *Concurrent Distinct Choices* pattern comes handy.

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
```

A letter of the cryptarithmetic puzzle is bound 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
```

The argument `token` must be unique for each letter, hence, it is natural and convenient to represent it with the letter itself. The store, identified by the variable `store`, is defined in the scope of the function. The operator `!!` applied to arguments l and i returns the i-th element of the list l.

The letters of the cryptarithmetic puzzle are computed as follows:

```
vs = nzdigit 'S'
ve = digit   'E'
vn = digit   'N'
...
```

where `nzdigit` is a variant of `digit` that returns only non-zero digits. For example, `(digit 'Y')` returns 2 if and only iff the second (starting from zero) element of the store is bound to `'Y'`. The entire program for this problem, *send-more.curry*, is referenced in Section 4.

The n-queens puzzle can be framed using the *Concurrent Distinct Choices* pattern, too, although concurrency is not a condition of this program. This program, *queenschoices.curry*, similar to many others for this problem, computes a permutation of the integers $0, 1, \ldots, n-1$, where the i-th element of the permutation is the row in which the queen in the i-th column is placed. A permutation can be seen as an injective mapping of the values $0, 1, \ldots, n-1$ into themselves. In this case, both the indexes and the values of this problem's mapping are most naturally represented by the integer numbers in the range 0 through $n-1$.

This pattern is not available in functional languages since they lack free variables. On the other hand, pure logic languages do not support concurrency by residuation (although some implementations offer coroutining) and the functional notation.

3.3 Incremental Solution

Name	*Incremental Solution*
Intent	compute solutions in an incremental manner
Applicability	a solution consists of a sequence of steps
Structure	non-deterministically extend a partial solution stepwise
Consequences	avoid explicit representation of the search space
Known uses	[27,37]
See also	often used with Constrained Constructor

A *solution* of a search problem is an element of a set, the search space, satisfying particular properties. To avoid both the enumeration of all the elements in the search space and the test of whether each element satisfies these properties, one defines a solution in an incremental manner, i.e., as a sequence of steps that extend a partial solution to a complete one. For instance, consider the problem, known as *stagecoach*, of constructing a path between two cities (see [29, p. 187]). The topology of a problem is represented by the connections between a set of cities using a `distance` function, e.g.:

```
distance Boston Chicago = 1500
distance Boston NewYork = 250
...
distance Denver LosAngeles = 1000
distance Denver SanFrancisco = 800
distance SanFrancisco LosAngeles = 300
```

An instance of this problem asks for a path from Boston to Los Angeles. A solution is a sequence of cities where Boston and Los Angeles are the first and last elements, respectively, and two consecutive elements are connected according to the `distance` function. Instead of this "monolithic" definition, it is preferable to define a solution in an incremental manner. A *partial solution* is any path from Boston to another city connected by `distance`. A complete solution is a partial solution with Los Angeles as the final element. A partial solution is extended, hopefully to a "more complete" one, by adding a new city reachable from the last one according to `distance`. It is convenient to represent a path as a list, in reverse order, of the cities of a partial solution. Extending a partial solution is implemented by the following function:

```
addCity (c:cs) | distance c c1 =:= d1
               = c1:c:cs                where c1,d1 free
```

Thus, a general search problem of this kind is specified by a triple: a function `extend` that extends a partial solution, the initial partial solution, and a predicate `complete` that defines the completeness of a partial solution. Based on such specification, the following non-deterministic search function computes a solution:

```
searchNonDet :: (ps->ps) -> ps -> (ps->Bool) -> ps
searchNonDet extend initial complete = solve initial
 where
   solve psol = if complete psol then psol
                                 else solve (extend psol)
```

The function `searchNonDet` is equivalent, except for the order of the arguments, to the function `until` found in the preludes of both Curry and Haskell. The Incremental Solution pattern greatly simplies the structure of the code in that no global search space is explicitly constructed. The function `searchNonDet` "sees" only a partial solution, i.e., a single path originating from the initial state. This works because the function `extend` is non-deterministic and the semantics of functional logic languages ensure the completeness of the computation.

The following expression computes a solution of our reachability problem:

```
searchNonDet addCity [Boston] (\(c:_)->c==LosAngeles)
```

The entire program, *stagecoach.curry*, implementing this problem is referenced in Section 4. This program has several advantages over a plausible formulation in a pure functional language. It is natural and convenient to define the stepwise extension of partial solutions as a non-deterministic function, see the definition of `addCity` and the examples of the Constrained Constructor pattern in Section 3.1. Non-deterministic specifications are simpler and more adaptable to new situations than equivalent deterministic specifications. For instance, the distance between two cities is defined as a (partial) function on cities which can be used in a narrowing-based functional logic language in a flexible way. To expand the topology of our sample problem with east-bound connections, it suffices to add the following symmetric rule to the definition of `addCity`:

```
addCity (c:cs) | distance c1 c =:= d1
               = c1:c:cs                where c1,d1 free
```

Although this expansion generates paths of unbounded length, successful connections are nevertheless found with the proposed search function if the implementation either evaluates all the alternatives in a fair manner, as in [6,26], or appropriately uses the *Constrained Constructor* pattern discussed earlier.

Among the advantages of this pattern is a modular architecture that separates the definition or specification of a problem from the computation of its solution. This separation makes different computations interchangeable. For instance, suppose that `searchDepthFirst` is a search strategy (its definition is shown in Appendix A) computing the possibly infinite list of the solutions of the stagecoach problem in a depth-first order. The list of solutions is obtained by evaluating:

```
searchDepthFirst addCity [Boston] (\(c:_)->c==LosAngeles)
```

We can refine the problem by including in a path its length which is defined as the sum of the distances of consecutive cities:

```
data DPath = DPath Int [City]

extendPath (DPath d (c:cs)) | distance c c1 =:= d1
                            = DPath (d+d1) (c1:c:cs)
                            where c1,d1 free

startAtBoston = DPath 0 [Boston]

reachedLA (DPath _ (c:_)) = c == LosAngeles
```

Now, we compute solutions containing distance information by evaluating the expression:

```
searchNonDet extendPath startAtBoston reachedLA
```

In order to compute the solutions with the shortest distance first, we define an appropriate comparison predicate between partial solutions:

```
shorter (DPath d1 _) (DPath d2 _) = d1<d2
```

Now, we compute the shortest path by applying a "best-first" strategy (its implementation is shown in Appendix A) to the problem:

```
searchBestFirst extendPath startAtBoston reachedLA shorter
```

Many other search problems are conveniently implemented by programs, e.g., *mission.curry*, *queensincr.curry* and *waterjug.curry*, that use the *Incremental Solution* pattern.

3.4 Locally Defined Global Identifier

Name	*Locally Defined Global Identifier*
Intent	ensure that a local name is globally unique
Applicability	a global identifier is declared in a local scope
Structure	introduce local names as logical variables to be bound later
Consequences	local names are globally unique
Known uses	Curry's GUI library [23] and HTML/CGI library [24]
See also	often used with Opaque Type

Lists and trees are ubiquitous datatypes in functional and logic programming because of their simplicity. In many application areas, the use of these simple datatypes leads to unnatural models of a problem resulting in error-prone programs that are difficult to maintain. For instance, graphical user interfaces can be considered tree-like structures since widgets can be grouped in containers that are used as widgets themselves. However, these structures may include dependencies among each other, e.g., a button widget may manipulate another widget in a different hierarchy. Thus, a graph structure is a more appropriate model in this situation. The usual representation of a graph as an algebraic type is based on the definition of a pair consisting of nodes and edges:

```
data Graph = Graph [Node] [Edge]
```

Edges consist (at a minimum) of a source and a target node, i.e., we need a unique identification of nodes in order to specify the edges between them. If we identify nodes by unique integers, we obtain:

```
data Node = Node Int

data Edge = Edge Int Int
```

Depending on the application, additional information items are included into both nodes and edges, e.g., lengths of edges, names of nodes, etc., that we omit for the sake of clarity. With these assumptions, a simple graph instance is:

```
g1 = Graph [Node 1, Node 2, Node 3]
           [Edge 1 2, Edge 3 2, Edge 1 3, Edge 3 3]
```

Unfortunately, graph instances of this kind cannot be composed and lack desirable properties like functional abstraction. For instance, if `addGraphs` is a function that composes two graphs by joining their nodes and edges, respectively, the expression (addGraphs g1 g1) produces a non-intended graph containing supposedly different nodes with identical numbers.

This problem is avoided by passing a counter through all the nodes when all the graphs are defined. However, this solution leads to code that is non-reusable and difficult to both understand and maintain because two separate tasks are interleaved.

The *Locally Defined Global Identifier* pattern elegantly solves this problem. This pattern separates the local definition of names from the task of assigning globally unique identifiers. The idea is to use unbound local variables as names when defining or creating graphs. Following this idea, we define the graph of the previous example as follows:

```
g1 = Graph [Node n1, Node n2, Node n3]
           [Edge n1 n2, Edge n3 n2, Edge n1 n3, Edge n3 n3]
   where n1,n2,n3 free
```

Since n1, n2 and n3 are local variables, g1 becomes compositional as a list or a tree would be. For example, (addGraphs g1 g1) is a graph with six different nodes.

To connect two graphs of this kind with an additional edge, one "exposes" the nodes intended for the connection:

```
g2 = (Graph [Node n1, Node n2, Node n3]
            [Edge n1 n2, Edge n3 n2, Edge n1 n3, Edge n3 n3],
      n1)
   where n1,n2,n3 free
```

The following function connects graph/node pairs with an edge provided that addEdge is a function that adds a new edge between two nodes of a graph:

```
connectGraphs (g,m) (h,n) = addEdge m n (addGraphs g h)
```

Now, (connectGraphs g2 g2) defines a graph consisting of six nodes and nine edges. The locally defined identifiers n1, n2 and n3 act as global identifiers in the composition.

Since unbound variables are not expressive, one may wish to instantiate them in some applications, e.g., visualization. To visualize graphs, one instantiates the node identifiers to pairwise distinct numbers or strings as usually required by visualization tools. The following function implements this process:

```
finalizeGraph (Graph ns es) = Graph (numberNodes 1 ns) es
  where numberNodes _ [] = []
        numberNodes n (Node ni : nodes)
            | ni =:= n  -- assign unique identifier
            = Node ni : numberNodes (n+1) nodes
```

A skeletal program, *graph.curry*, showing the use of this pattern is referenced in Section 4.

This pattern is not only useful in graph-based applications, but also in applications where hierarchical (tree-like) data structures are appropriate, but additional references inside such structures are needed. As mentioned earlier, graphical user interfaces are one class of such applications. This pattern has been applied in this context in [23]. Another application area is dynamic web page generation with form-based input. HTML documents are structured as trees, but the input forms and their “submit” buttons contain dependencies between subtrees that can be appropriately described with this pattern [24].

The `Graph` datatype is only a simple example demonstrating the basic use of this pattern. In real world applications, this pattern is refined in various ways. For instance, the use of logical variables instead of concrete numbers in the definition of graphs is only a guideline for the programmer. By contrast, the *Opaque Type* pattern presented in the next section enforces the use of logical variables exclusively. A remaining problem not addressed by these patterns is ensuring that the variables used in different nodes are distinct. This situation occurs in other environments as well, e.g., Tcl/Tk or Perl/CGI programs, which use additional analysis to solve the problem. In functional logic programming, a convenient option is using the *Constrained Constructor* pattern discussed earlier.

This pattern is not available in functional languages since they lack free variables. As a consequence, functional approaches to GUI or HTML programming use a more imperative style and/or lack compositionality [11,33]. Erwig [13] proposes an inductive definition of graphs that supports coding graph algorithms in a functional style. His approach is specific to graphs and does not lead to appropriate descriptions of the GUI and HTML applications that we mentioned.

3.5 Opaque Type

Name	*Opaque Type*
Intent	ensure that values of a datatype are hidden
Applicability	define instances of a type whose values are unknown
Structure	wrap values with a private constructor
Consequences	values can only be denoted by free variables
Known uses	Curry’s GUI library [23] and HTML/CGI library [24]
See also	

In applications containing elements which are interesting only in relation to each other, it is often desirable to hide the values of these elements since they are either irrelevant or are computed by some function of the application. The construction of graphical user interfaces and interactive HTML documents mentioned in the previous section and abstracted by graphs are examples of this situation. A problem of this situation is that the programmer may have to construct instances of a type, but no value of that type is available.

In the graph example of the previous section, the values of the datatype that abstracts node identifiers should be hidden, but the datatype itself should be visible to construct the nodes. A convenient option to satisfy this condition is the use of unbound variables rather than literal values for node identifiers when a graph is defined. As an added bonus, this condition gives the implementor of a graph library the freedom to change this datatype, e.g., from `Int` to `String` whenever it is convenient, without affecting a client of the library.

The values of a datatype can be hidden by "wrapping" them with a private constructor of the datatype. This means that literal values are replaced by values of an abstract datatype that has no public constructors. The values of this datatype can be denoted by unbound variables.

For instance, consider the graph of Section 3.4. To hide the use of integers to identify nodes, we define in the graph library a datatype for *node identifiers*:

```
data NodeId = NodeId Int
```

where the constructor `NodeId` is not exported—a standard feature of module systems. Furthermore, we change the definition of nodes and edges so that we use the type `NodeId` wherever nodes are required:

```
data Node = Node NodeId
data Edge = Edge NodeId NodeId
```

These definitions are in the same module that declares `NodeId` and consequently may access it even though it is private. Finally, we adapt all the functions in the graph library where node identifiers are involved. These functions may access `NodeId` as well. In our example we slightly change the definition of `finalizeGraph` by replacing "`ni =:= n`" with "`ni =:= NodeId n`." This change is completely invisible to the user of the library. The coding of graphs remain identical, but the pattern ensures that the arguments of `Node` are exclusively unbound variables.

Program *graph.curry*, mentioned in the previous section and referenced in Section 4, shows an application of this pattern as well.

This pattern is not directly available in functional languages since they lack free variables. Although most functional languages have a module system that allows the programmer to hide values, using hidden values require accessor functions that may be more difficult to define when the values are distributed across tree-structured datatypes.

4 Exemplary Programs

The programs discussed in this paper are available at URL:

```
http://www.cs.pdx.edu/~antoy/flp/patterns/
```

5 Conclusion

We have presented five software design patterns specifically intended for a functional logic language. Our patterns are quite general. In the short programs referenced in Section 4, we find repeated opportunities of applications of our patterns. In several cases, more than one pattern is appropriately used in the same program.

This is the first paper on patterns for a functional logic language. Therefore, the patterns that we have selected for our small catalog address essential activities of program design and implementation. The *Constrained Constructor* offers a technique for using types that are more specialized than those directly available in functional and functional logic languages. The *Concurrent Distinct Choices* supports a simple and efficient implementation of an injective mapping where the mapping of indexes to values may occur concurrently or in no pre-established order. The *Incremental Solution* suggests a flexible general architecture for non-deterministic search problems which avoids the explicit manipulation of a global search space. The *Locally Defined Global Identifier* and the *Opaque Type* are intended for the definition of respectively global identifiers in local scopes and instances of a type whose values are hidden. These patterns simplify data construction in a way that promotes code modularity and reuse.

Object-oriented patterns are classified according to tasks, e.g., creational patterns for the construction of objects, behavioral patterns for the execution of code, etc., and develop some general themes, e.g., replacement of inheritance, a static property, with delegation, a dynamic property. Our catalog is yet too small for meaningful classifications, but it already outlines two general themes. The theme of the first three patterns that we presented is the use of non-determinism in computations. The theme of the last two patterns is the use of logical variables in expressions.

The elegance of some of our patterns and the ease with which they solve some difficult problems highlight the features that distinguish a functional logic language from other paradigms. Of course, the functional evaluation is an essential aspect since it provides sophisticated efficient control of execution, through lazy evaluation, in a purely declarative manner, e.g., without the Prolog "cut." A related important aspect is non-determism, specifically the integration of non-determinism with functional evaluation. This combination supports implicit search in the logic programming style without sacrificing the efficient control of execution discussed earlier. Another essential aspect are logic variables, specifically the integration of logic variables with functional evaluation. This combination, made possible by narrowing, supports both equational reasoning and functional inversion in the logic programming style. The final aspect is concurrency, specifically the interleaving of different computations. The interleaving of deterministic and non-deterministic computations may reduce the size of the search space and consequently improve the overall efficiency of a program.

Our patterns rely on these essential aspects of functional logic languages. Consequently, they can be used in any functional logic language that supports these characterizing aspects. Curry supports all these aspects. For this reason, and for the availability of an efficient, robust and fairly complete implementation, PAKCS, we have chosen Curry as the presentation language.

Industry is showing a growing interest in programming with patterns. The proved or perceived benefits of using patterns for software development include clarity of design, faster development, lower costs, robustness, efficiency and ease of maintenance. Although most often the programming language used in industry is object oriented, patterns are more concerned with ideas than with code and many benefits of using patterns are largely independent of the language.

We plan to continue maintaining our on-line catalog and to add new functional logic patterns as they become recognized and documented.

References

1. H. Aït-Kaci. An overview of LIFE. In J. Schmidt and A. Stogny, editors, *Proc. Workshop on Next Generation Information System Technology*, pages 42–58. Springer LNCS 504, 1990.

2. H. Aït-Kaci, P. Lincoln, and R. Nasr. Le Fun: Logic, equations, and functions. In *Proc. 4th IEEE Internat. Symposium on Logic Programming*, pages 17–23, San Francisco, 1987.
3. S. Antoy. Optimal non-deterministic functional logic computations. In *Proc. International Conference on Algebraic and Logic Programming (ALP'97)*, pages 16–30. Springer LNCS 1298, 1997.
4. S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *Journal of the ACM*, 47(4):776–822, 2000.
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, M. Hanus, B. Massey, and F. Steiner. An implementation of narrowing strategies. In *Proc. of the 3rd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP 2001)*, pages 207–217. ACM Press, 2001.
7. K. Beck and W. Cunningham. Using pattern languages for object-oriented programs. In *Specification and Design for Object-Oriented Programming (OOPSLA-87)*, 1987.
8. J. Boye. S-SLD-resolution – an operational semantics for logic programs with external procedures. In *Proc. of the 3rd Int. Symposium on Programming Language Implementation and Logic Programming*, pages 383–393. Springer LNCS 528, 1991.
9. F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla. The Ciao Prolog system. Reference manual. Technical report CLIP3/97.1, School of Computer Science, Technical University of Madrid (UPM), 1997.
10. R. Caballero and F. López-Fraguas. A functional-logic perspective of parsing. In *Proc. of the 4th Fuji Int'l Symposium on Functional and Logic Programming*, pages 85–99, Tsukuba, Japan, 1999. Springer LNCS 1722.
11. K. Claessen, T. Vullinghs, and E. Meijer. Structuring graphical paradigms in TkGofer. In *Proc. of the International Conference on Functional Programming (ICFP'97)*, pages 251–262. ACM SIGPLAN Notices Vol. 32, No. 8, 1997.
12. J. W. Cooper. *Java Design Patterns*. Addison Wesley, 2000.
13. M. Erwig. Functional programming with graphs. In *2nd ACM SIGPLAN Int. Conf. on Functional Programming (ICFP'97)*, pages 52–65, 1997.
14. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.
15. J. González-Moreno, M. Hortalá-González, F. López-Fraguas, and M. Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming*, 40:47–87, 1999.
16. M. Grand. *Patterns in Java*. J. Wiley, 1998.
17. M. Hanus. Improving control of logic programs by using functional logic languages. In *Proc. of the 4th International Symposium on Programming Language Implementation and Logic Programming*, pages 1–23. Springer LNCS 631, 1992.
18. M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994.
19. M. Hanus. Analysis of residuating logic programs. *Journal of Logic Programming*, 24(3):161–199, 1995.
20. M. Hanus. Efficient translation of lazy functional logic programs into Prolog. In *Proc. Fifth International Workshop on Logic Program Synthesis and Transformation*, pages 252–266. Springer LNCS 1048, 1995.
21. M. Hanus. A unified computation model for functional and logic programming. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages (Paris)*, pages 80–93, 1997.

22. M. Hanus. Distributed programming in a multi-paradigm declarative language. In *Proc. of the International Conference on Principles and Practice of Declarative Programming (PPDP'99)*, pages 376–395. Springer LNCS 1702, 1999.
23. 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.
24. 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.
25. M. Hanus, S. Antoy, 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/`, 2002.
26. 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.
27. M. Hanus and F. Steiner. Controlling search in declarative programs. In *Principles of Declarative Programming (Proc. Joint International Symposium PLILP/ALP'98)*, pages 374–390. Springer LNCS 1490, 1998.
28. M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.7). Available at `http://www.informatik.uni-kiel.de/~curry`, 2000.
29. E. Horowitz and S. Sahni. *Fundamentals of Computer Algorithms*. Computer Science Press, 1978.
30. J. Langr. *Essential Java Style: Patterns for Implementation*. Prentice Hall, 2000.
31. J. Lloyd. Programming in an integrated functional and logic language. *Journal of Functional and Logic Programming*, (3):1–49, 1999.
32. F. López-Fraguas and J. Sánchez-Hernández. TOY: A Multiparadigm Declarative System. In *Proc. of RTA'99*, pages 244–247. Springer LNCS 1631, 1999.
33. E. Meijer. Server side web scripting in Haskell. *Journal of Functional Programming*, 10(1):1–18, 2000.
34. L. Naish. Adding equations to NU-Prolog. In *Proc. of the 3rd Int. Symposium on Programming Language Implementation and Logic Programming*, pages 15–26. Springer LNCS 528, 1991.
35. S. Peyton Jones and J. Hughes. Haskell 98: A non-strict, purely functional language. `http://www.haskell.org`, 1999.
36. V. Saraswat. *Concurrent Constraint Programming*. MIT Press, 1993.
37. C. Schulte and G. Smolka. Encapsulated search for higher-order concurrent constraint programming. In *Proc. of the 1994 International Logic Programming Symposium*, pages 505–520. MIT Press, 1994.
38. G. Smolka. The Oz programming model. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, pages 324–343. Springer LNCS 1000, 1995.
39. Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1-3):17–64, 1996.

A Further Search Strategies for the Incremental Solution Pattern

In this appendix we show the implementation of additional search strategies to compute solutions to problems suitable for the *Incremental Solution* pattern. These strategies are variations or refinements of the *Incremental Solution* in the sense that they compute solutions by incremental steps.

First, we show the implementation of a *depth-first* strategy which enumerates the possibly infinite list of all the solutions in a depth-first order. We make use of the predefined operation `findall` for encapsulating search [27]. The expression `findall \`x`->`c computes the list of all the solutions for x w.r.t. a constraint c. The depth-first strategy is implemented by:

```
searchDepthFirst :: (ps->ps) -> ps -> (ps->Bool) -> [ps]
searchDepthFirst extend initial complete = solve [initial]
    where
        solve []       = []
        solve (st:sts) = if complete st
                             then st : solve sts
                             else solve (expand (st:sts))

        nextstates st = findall \x -> extend st =:= x

        expand (st:sts) = nextstates st ++ sts
```

A *breadth-first* search strategy is similarly defined by swapping the order of the concatenation of the partial solutions in the local function `expand`. This has the advantage that solutions are computed even in the presence of infinite expansions of partial solutions. On the other hand, it requires more memory to store all intermediate partial solutions.

A *best-first* search strategy that expands the "best" partial solutions first requires an additional parameter. This parameter is a predicate, `better`, that determines which of two partial solutions is better according to an arbitrary general criterion. Namely, "`better s1 s2`" where `s1` and `s2` are partial solutions is intended to be true if `s1` is better than `s2`. The definition of this search strategy is similar to `searchDepthFirst`, the difference being that the list of partial solutions that are candidate for expansion is sorted according to `better`. The best first strategy is implemented by:

```
searchBestFirst :: (ps->ps) -> ps ->
                   (ps->Bool) -> (ps->ps->Bool) -> [ps]
searchBestFirst extend initial complete better
    = solve [initial]
    where
        solve []       = []
        solve (st:sts) = if complete st
                             then st : solve sts
                             else solve (expand (st:sts))

        nextstates st = findall \x -> extend st =:= x

        expand (st:sts) =
            merge (sort better(nextstates st)) sts

        merge []          sts         = sts
        merge (st:sts)    []          = st:sts
        merge (st1:sts1) (st2:sts2) =
          if better st1 st2 then st1 : merge sts1 (st2:sts2)
                            else st2 : merge (st1:sts1) sts2
```