New Functional Logic Design Patterns

Sergio Antoy[1] Michael Hanus[2]

[1] Computer Science Dept., Portland State University, Oregon, U.S.A.
antoy@cs.pdx.edu

[2] Institut für Informatik, CAU Kiel, D-24098 Kiel, Germany.
mh@informatik.uni-kiel.de

Abstract. Patterns distill successful experience in solving common software problems. We introduce a handful of new software design patterns for functional logic languages. Some patterns are motivated by the evolution of the paradigm in the last 10 years. Following usual approaches, for each pattern we propose a name and we describe its intent, applicability, structure, consequences, etc. Our patterns deal with fundamental aspects of the design and implementation of functional logic programs such as function invocation, data structure representation and manipulation, specification-driven implementation, pattern matching, and nondeterminism. 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, the *Constrained Constructor* pattern [3], expresses the idea of calling a data constructor exclusively indirectly through an intermediate function to avoid undesirable instances of some type. The idea is applicable to a variety of problems, but the code of the intermediate function is dependent on each problem.

Patterns originated from the development of object-oriented software [6] and became both a popular practice and an engineering discipline after [11]. As the landscape of programming languages evolves, patterns are "translated" from one language into another [10, 12]. 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 [11], which solves the problem of adapting a service to a client coded for different interface, is language independent. The *Facade* pattern [11], which

To appear in the Proceedings of the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011)
Odense, Denmark, Springer LNCS, Vol. 6816, pages 19-34, 2011
The original publication is available at www.springerlink.com

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 *Visitor* pattern [11], which enables extending the functionality of a class without modifying the class interface, is critically dependent on features of object orientation, such as overloading and overriding.

Patterns are related to both idioms and pearls. Patterns are more articulated than idioms, which never cross languages boundaries, and less specialized than pearls, which often are language specific. The boundaries of these concepts are somewhat arbitrary. Patterns address general structural problems and therefore we use this name for our concepts.

Patterns for a declarative paradigm—in most cases specifically for a functional logic one—were introduced in [3]. This paper is a follow up. Ten years of active research in functional logic programming have brought new ideas and deeper understanding, and in particular some new features and constructs, such as functional patterns [4] and set functions [5]. Some patterns presented in this paper originates from these developments.

High-level languages are better suited for the implementation of reusable code than imperative languages, see, e.g., parser combinators [7]. Although whenever possible 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 primary emphasis is not on efficiency, but on clarity and simplicity of design and ease of understanding and maintenance. Interestingly enough, one of our patterns is concerned with moving from the primary emphasis to more efficient code. Our presentation of a pattern follows the usual (metapattern) approaches that provide, e.g., name, intent, applicability, structure, consequences, etc. Some typical elements, such as "known uses," are sparse or missing because functional logic programming is a still relatively young paradigm. Work on patterns for this paradigm is slowly emerging.

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 concludes the paper.

2 Functional Logic Programming and Curry

A Curry program is a set of functions and data type definitions in Haskell-like syntax [26]. Data type definitions have the same semantics as Haskell. A function is defined by conditional rewrite rules of the form:

$$f\ t_1 \ldots t_n \ \texttt{|}\ c\ \texttt{=}\ e \quad \texttt{where}\ vs\ \texttt{free} \tag{1}$$

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 offers two main features for logic programming: logical (free) variables and non-deterministic functions. Logical variables are declared by a "`free`" clause as shown above, can occur in conditions and/or

right-hand sides of defining rules, and are instantiated by narrowing [1, 2], a computation similar to resolution, but applicable to functions of any type rather than predicates only. Similarly to Haskell, the "`where`" clause is optional and can also contain other local function and/or pattern definitions.

Non-deterministic functions are defined by overlapping rules such as:

```
(?) :: a -> a -> a
x ? y = x
x ? y = y
```

In contrast to Haskell, in which the first matching rule is applied, in Curry all matching (to be more precise, unifiable) rules are applied—non-deterministically. For example, `0?1` has two values, `0` and `1`. The programmer has no control over which value is selected during an execution, but will typically constrain this value according to the intent of the program. In particular, 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. By contrast, the operator "`&&`" denotes the usual Boolean conjunction which evaluates to either `True` or `False`.

An example of the features discussed above can be seen in the definition of a function that computes the last element of a non-empty list. The symbol "`++`" denotes the usual list concatenation function:

```
last l | p++[e]=:=l = e   where p,e free
```

As in Haskell, the rules defining most functions are *constructor-based* [25], in (1) $t_1 \ldots t_n$ are made of variables and/or data constructor symbols only. However, in Curry we can also use a *functional pattern* [4]. With this feature, which relies on narrowing, we can define the function `last` also as:

```
last (_++[e]) = e
```

The operational semantics of Curry, precisely described in [14, 22], 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 Curry is based on an optimal evaluation strategy [2], it can be considered a generalization of concurrent constraint programming [27] with a lazy strategy.

Furthermore, Curry also offers features for application programming like modules, monadic I/O, encapsulated search [21], ports for distributed programming [15], libraries for GUI [16] and HTML programming [17], etc. We do not present these aspects of the language, since they are not necessary for understanding our contribution. There exist several implementations of Curry. The examples presented in this paper were all compiled and executed by PAKCS [20], a compiler/interpreter for a large subset of Curry.

There exist also other functional logic languages, most notably $\mathcal{TOY}$ [8, 24], with data types, possibly non-deterministic functions, and logic variables instantiated by narrowing similar to Curry. Many patterns and exemplary programs

discussed in this paper are adaptable to these languages with minor, often purely syntactic, changes.

3 Patterns

In this section we present a small catalog of patterns that address non-trivial solutions of some general and challenging problems.

3.1 Call-by-reference

Name	*Call-by-reference*
Intent	return multiple values from a function without defining a containing structure
Applicability	a function must return more than one value
Structure	an argument passed to a function is an unbound variable
Consequences	avoid constructing a structure to hold multiple values
Known uses	Parser combinators
See also	Monads, Extensions

The pattern name should not mislead the reader. There is no *call-by-reference* in functional logic languages. The name stems from a similarity with the passing mode in that a value is returned by a function through an argument of the call.

When a function must return multiple values, a standard technique is to return a structure that holds all the values to be returned. For example, if function f must return both a value of type A and a value of type B, the return type could be (A, B), a pair with components of type A and B, respectively. The client of f extracts the components of the returned structure and uses them as appropriate. Although straightforward, this approach quickly becomes tedious and produces longer and less readable code. This pattern, instead, suggests to pass unbound variables to the function which both returns a value and binds other values to the unbound variables.

Example: A *symbol table* is a sequence of records. A *record* is a pair in which the first component is intended as a key mapped to the second component.

```
type Record = (String,Int)
type Table = [Record]
```

The function `insert` attempts to insert a record (k, v) into a table t which is expected to contain no record with key k. This function computes both a Boolean value, `False` if a record with key k is already in t, `True` otherwise, and the updated table, if no record with key k is in t. Attempting to insert a record whose key is in the table is an error, hence the returned table in this case is uninteresting. The Boolean value is returned by the function whereas the updated table is bound to the third argument of the call. Alternatively, the function could return the updated table and bind the Boolean value to the third argument, but as we will discuss shortly this option is not as appealing.

```
insert :: Record -> Table -> Table -> Bool
insert (k,v) [] x = x =:= [(k,v)] &> True
insert (k,v) ((h,w):t) x
  | k == h = x =:= (h,w):t &> False
  | otherwise = let b = insert (k,v) t t'
                in x =:= (h,w):t' &> b  where t' free
```

The operator "`&>`", called *constrained expression*, takes a constraint as its first argument. It solves this constraint and, if successful, returns its second argument.

The function `remove` attempts to remove a record with key k from a table t which is expected to contain one and only one such record. This function computes both a Boolean value, `True` if a record with key k is in t, `False` otherwise, and the updated table if a record with key k is in t. Attempting to remove a record whose key is not in the table is an error, hence the returned table in this case is uninteresting.

```
remove :: String -> Table -> Table -> Bool
remove _ [] [] = False
remove k ((h,w):t) x
  | k == h = x =:= t &> True
  | otherwise = let b = remove k t t'
                in x =:= (h,w):t' &> b  where t' free
```

An example of use of the above functions follow, where the key is a string and the value is an integer.

```
emptyTable = []
test = if insert ("x",1) emptyTable t1 &&
          insert ("y",2) t1 t2 &&
          remove "z" t2 t3 then t3
       else error "Oops"
       where t1, t2, t3 free
```

Of the two values returned by functions `insert` and `remove`, the table is subordinate to the Boolean in that when the Boolean is false, the table is not interesting. This suggest returning the Boolean from the functions and to bind the table to an argument. A client typically will test the Boolean before using the table, hence the test will trigger the binding. However, variables are bound only to fully evaluated expressions. This consideration must be taken into account to select which value to return in a variable when laziness is crucial.

For a client, it is easier to use the functions when they are coded according to the pattern rather than when they return a structure. A *state monad* [23] would be a valid alternative to this pattern for the example presented above and in other situations. Not surprisingly, this pattern can be used instead of a *Maybe* type.

This pattern is found, e.g., in the parser combinators of [7]. A parser with representation takes a sequence of tokens and typically a free variable, which is bound to the representation of the parsed tokens, whereas the parser returns the

sequence of tokens that remain to be parsed. The *Extensions* of [9] are a form of this pattern. The reference contains a comparison with the monadic approach.

This pattern is not available in functional languages since they lack free variables. Logic languages typically return information by instantiating free variables passed as arguments to predicates, but predicates do not return information, except for succeeding.

3.2 Many-to-many

Name	*Many-to-many*
Intent	encode a many-to-many relation with a single simple function
Applicability	a relation is computed in both directions
Structure	a non-deterministic function defines a one-to-many relation; a functional pattern defines the inverse relation
Consequences	avoid structures to define a relation
Known uses	
See also	

We consider a many-to-many relation $\mathcal{R}$ between two sets A and B. Some element of A is related to distinct elements of B and, vice versa, distinct elements of A are related to some element of B. In a declarative program, such a relation is typically abstracted by a function f from A to subsets of B, such that $b \in f(a)$ iff $a \mathcal{R} b$. We will call this function the *core function* of the relation. Relations are dual to graphs and, accordingly, the core function can be defined, e.g., by an adjacency list. The relation $\mathcal{R}$ implicitly defines an inverse relation which, when appropriate, is encoded in the program by a function from B to subsets of A, the core function of the inverse relation.

In this pattern, the core function is encoded as a non-deterministic function that maps every $a \in A$ to every $b \in B$ such that $a \mathcal{R} b$. The rest of the abstraction is obtained nearly automatically using standard functional logic features and libraries. In particular, the core function of the inverse relation, when needed, is automatically obtained through a functional pattern. The sets of elements related to a given element are automatically obtained using the set functions of the core function.

Example: Consider an abstraction about blood transfusions. We define the blood types and the function `giveTo`. The identifiers `Ap`, `An`, etc. stand for the types A *positive* ($A+$), A *negative* ($A-$), etc. The application `giveTo` x returns a blood type y such that x can be given to a person with type y. E.g., $A+$ can be given to both $A+$ and $AB+$.

```
data BloodType = Ap | An | ABp | ABn | Op | On | Bp | Bn

giveTo :: BloodType -> BloodType
giveTo Ap  = Ap ? ABp
giveTo Op  = Op ? Ap ? Bp ? ABp
giveTo Bp  = Bp ? ABp
...
```

The inverse relation is trivially obtained with a function defined using a functional pattern [4]. The application `receiveFrom` x returns a blood type y such that a person with type x can receive type y. E.g., $AB+$ can receive $A+$, $AB+$ and $O+$ among others.

```
receiveFrom :: BloodType -> BloodType
receiveFrom (giveTo x) = x
```

To continue the example, let us assume a database defining the blood type of a set of people, such as:

```
has :: String -> BloodType
has "John" = ABp
has "Doug" = ABn
has "Lisa" = An
```

The following function computes a donor for a patient, where the condition $x \neq y$ avoids self-donation, which obviously is not intended.

```
donorTo :: String -> String
donorTo x
   | giveTo (has y) =:= has x & x =/= y
   = y
   where y free
```

E.g., the application `donorTo "John"` returns both `"Doug"` and `"Lisa"`, whereas `donorTo "Lisa"` correctly fails for our very small database.

To continue the example further, we may need a particular set of donors, e.g., all the donors that live within a certain radius of a patient and we may want to rank these donors by the date of their last blood donation. For these computations, we use the set function [5] automatically defined for any function. The function `donorTo'set` produces the set of all the donors for a patient. The *SetFunctions* library module offers functions for filtering and sorting this set.

Many-to-many relations are ubiquitous, e.g., students taking courses from teachers, financial institutions owning securities, parts used to build products, etc. Often, it won't be either possible or convenient to hard-wire the relation in the program as we did in our example. In some cases, the core function of a relation will access a database or some data structure, such as a search tree, obtained from a database. An interesting application of this pattern concerns the relation among the functions of a program in which a function is related to any function that it calls. In this case, we expect that the compiler will produce a structure, e.g., a simple set of pairs, which the core function will access for its computations. Beside a small difference in the structure of the core function, the rest of the pattern is unchanged.

This pattern is not available in functional languages since they lack non-deterministic functions. Logic languages support key aspects of this pattern, in particular, the non-determinism of the core function and the possibility of computing a relation and its inverse relation with the same predicate.

3.3 Quantification

Name	*Quantification*
Intent	encode first-order logic formula in programs
Applicability	problems specified in a first-order logic language
Structure	apply "*there exists*" and "*for all*" library functions
Consequences	programs are encoded specifications
Known uses	
See also	

First-order logic is a common and powerful language for the specification of problems. The ability to execute even some approximation of this language enables us to directly translate many specifications into programs. A consequence of this approach is that the logic of the resulting program is correct by definition and the code is obtained with very little effort. The main hurdle is existential quantification, since specifications of this kind are often not constructive. However, narrowing, which is the most characterizing feature of functional logic languages, supports this approach.

Narrowing evaluates expressions, such as a constraint, containing free variables. The evaluation computes some instantiations of the variables that lead to the value of the expression, e.g., the satisfaction of the constraint. Hence, it solves the problem of existential quantification.

Universal quantification is more straightforward. Mapping and/or folding operations on sets are sufficient to verify whether all the elements of the set satisfy some condition. In particular, set functions can be a convenient means to compute the sets required by an abstraction.

We define the following two functions for existential and universal quantification, where `Values` is a library-defined polymorphic type abstracting a set and `mapValues` and `foldValues` are standard mapping and folding functions on sets. The function `exists` is a simple idiom defined only to improve the readability of the code.

```
exists :: a -> (a -> Success) -> Success
exists x f = f x

forall :: Values a -> (a -> Bool) -> Success
forall s f = foldValues (&&) True (mapValues f s) =:= True
```

Example: Map coloring is stated as "given any separation of a plane into contiguous regions, producing a figure called a map, ... color the regions of the map so that no two adjacent regions have the same color" [28]. A map coloring problem has a solution M iff there exists a colored map M such that for all x and y regions of M and x adjacent to y the colors of x and y differ. The above statement is a specification of the problem stated semi-formally in a first-order logic language.

We begin by defining the regions of the map and the adjacency relation. For the curious, the map is the Pacific North West.

```
data State = WA | OR | ID | BC

states = [WA,OR,ID,BC]

adjacent = [(WA,OR),(WA,ID),(WA,BC),(OR,ID),(ID,BC)]
```

To continue the example, we define the colors to use for coloring the map, only 3, and the function that colors a state. Coloring a state is a non-deterministic assignment, represented as a pair, of a color to a state.

```
data Color = Red | Green | Blue

color :: a -> (a,Color)
color x = (x, Red ? Green ? Blue)
```

The rest of the program follows:

```
solve :: [(State,Color)]
solve | exists cMap (\map ->
           forall someAdj'set (\(st1,st2) ->
              lookup st1 map /= lookup st2 map))
      = cMap
      where cMap = map color states
            someAdj = foldr1 (?) adjacent
```

The identifier `cMap` is bound to some colored map. The identifier `someAdj` is bound to some pair of adjacent states. The identifier `someAdj'set` is bound to the implicitly defined set function of `someAdj`, hence it is the set of all the pairs of adjacent states. The function `lookup` is defined in the standard *Prelude*. It retrieves the color assigned to a state in the colored map.

The condition of the function `solve` is an encoded, but verbatim, translation of the specification. The condition could be minimally shortened by eliminating the `exists` idiom, but the presented form is more readable and shows the pattern in all its generality.

Since $\forall x\ P$ is equivalent to $\neg\exists x\ \neg P$, we also define:

```
notExists :: Values a -> (a -> Bool) -> Success
notExists s f = foldValues (||) False (mapValues f s) =:= False
```

This pattern is very general and applicable to problems, whether or not deterministic, which have non-constructive specifications. For example, the minimum element of a collection can be specified as m is the minimum of C iff there exists some m in C such that there not exists some x in C such that $x < m$, i.e., $\exists m\,(m \in C \wedge \neg\exists x\,(x \in C \wedge x < m))$ or, equivalently, for all x in C, $x \geq m$

This pattern is not available in functional languages since they lack narrowing. Logic languages have some forms of existential quantification, but their lack of functional nesting prevents the readable and elegant notion available in functional logic languages.

3.4 Deep selection

Name	*Deep selection*
Intent	pattern matching at arbitrary depth in recursive types
Applicability	select an element with given properties in a structure
Structure	combine a type generator with a functional pattern
Consequences	separate structure traversal from pattern matching
Known uses	HTML and XML applications coded in Curry
See also	Curry's HTML library

Pattern matching is undoubtedly a convenient feature of modern declarative languages because it allows to easily retrieve the components of a data structure such as a tuple. Recursively defined types, such as lists and trees, have components at arbitrary depths that cannot be selected by pattern matching because pattern matching selects components only at predetermined positions. For recursively defined types, the selection of some element with a given property in a data structure typically requires code for the traversal of the structure which is intertwined with the code for using the element. The combination of functional patterns with type generators allows us to select elements arbitrarily nested in a structure in a pattern matching-like fashion without explicit traversal of the structure and mingling of different functionalities of a problem.

A list is a recursively defined type that can be used to represent a mapping by storing key-value pairs. One such structure, bound to the variable `cMap`, was used in the example of the *Quantification* pattern. The library function `lookup` retrieves from the mapping the value v associated to a key k. In that example, there is one and only one such pair in the list. The function `lookup` must both traverse the list to find the key and through pattern matching return the associated value. The two computations are intermixed and pattern matching some element with different characteristic in a list would require duplication of the code to traverse the list. Functional patterns offer a new option.

The key idea of the *deep selection* pattern is to define a "generator" function that generates all the instances of a type with a given element. This function is interesting for recursively defined types. For a list, this generator is:

```
withElem :: a -> [a]
withElem e = e:unknown ? unknown:withElem e
```

The function `unknown` is defined in the *Prelude* and simply returns a free variable. This generator supports the following definition of `lookup` in which the (functional) pattern is as simple as it can be.

```
lookup :: :: [(a,b)] -> b
lookup (withElem (_,v)) = v
```

The counterpart of `withElem` is `elemOf`, an "extractor" as opposed to a generator, which returns non-deterministically a component of a structure:

```
elemOf :: [a] -> a
elemOf (withElem e) = e
```

We will use both these functions including specialized variations of them.

Example: Below, we show a simple type for representing arithmetic expressions and a generator of all the expressions with a given subexpression:

```
data Exp = Lit Int
         | Var [Char]
         | Add Exp Exp
         | Mul Exp Exp

withSub :: Exp -> Exp
withSub exp = exp
            ? op (withSub exp) unknown
            ? op unknown (withSub exp)
   where op = Add ? Mul
```

Suppose that we want to find all the variables of an expression. The function `varOf`, a specialization of `elemOf` shown earlier, for the type `Exp`, takes an expression *exp* and returns the identifier of some variable of *exp*.

```
varOf :: Exp -> String
varOf (withSub (Var v)) = v
```

The set of identifiers of all the variables of *exp* is simply obtained with the set function of `varOf`, i.e., `varOf'set` *exp*.

In some situations, a bit more machinery is needed. For example, suppose that we want to find common subexpressions of an expression, such as `42` and `y` in the following:

```
Add (Mul (Lit 42) (Add (Lit 42) (Var "y")))
    (Add (Var "x") (Var "y"))
```

One option is a more specialized generator that generates all and only the expressions with a given common subexpression:

```
withCommonSub :: Exp -> Exp
withCommonSub exp = op (withCommonSub exp) unknown
                  ? op unknown (withCommonSub exp)
                  ? op (withSub exp) (withSub exp)
  where op = Add ? Mul
```

Another option is a different more specialized generator that generates all the expressions with a given subexpression at a given position. The position is a string of 1's and 2's defining a path from the root of the expression to the subexpression.

```
withSubAt :: [Int] -> Exp -> Exp
withSubAt [] exp = exp
withSubAt (1:ps) exp = (Add ? Mul) (withSubAt ps exp) unknown
withSubAt (2:ps) exp = (Add ? Mul) unknown (withSubAt ps exp)
```

This generator is useful to pattern match a subexpression and its position:

```
subAt :: Exp -> ([Int],Exp)
subAt (withSubAt p exp) = (p,exp)
```

In the new version of the function that computes a common subexpression, not only we return the common subexpression, but also the two positions at which subexpression occurs, since they are available. The ordering operator "`<:`" is predefined for all types. Its use in our code ensures that the same subexpression is not matched twice.

```
commonSub :: Exp -> (Exp,[Int],[Int])
commonSub exp | p1 <: p2 & e1=:=e2 = (e1,p1,p2)
              where (p1,e1) = subAt exp
                    (p2,e2) = subAt exp
```

This pattern is applied in HTML processing. Curry provides a library for the high-level construction of type-safe HTML documents and web-oriented user interfaces [18]. HTML documents are instances of a type `HtmlExp`, shown below, consisting of sequences of text and tag elements with both attributes and possibly nested elements.

```
data HtmlExp = HtmlText String
             | HtmlStruct String [(String,String)] [HtmlExp]
```

The problem, sought-after by spammers, of finding all the e-mail addresses in a HTML page is trivialized by this pattern. The following function finds some e-mail address in a document:

```
eAddress :: HtmlExp -> String
eAddress (withHtmlElem
            (HtmlStruct _
               (withElem ("href","mailto:"++name)) _)) = name
```

where `withElem`, defined above, is applied to match a `href` tag with value `mailto` in a list of attributes and the type generator `withHtmlElem`, defined below, is applied to match an `HtmlStruct` structure with the desired attribute in a tree of HTML structures.

```
withHtmlElem :: HtmlExp -> HtmlExp
withHtmlElem helem = helem
                   ? HtmlStruct unknown
                                unknown
                                (withElem (withHtmlElem helem))
```

All the addresses in a page are produced by the set function of `eAddress`.

In a similar way, one can also define generators for deep matching in XML structures. A library to support the declarative processing of XML data based on the deep selection pattern is described in [19].

This pattern is not available in both functional and logic languages since they lack functional patterns.

3.5 Non-determinism introduction and elimination

Name	*Non-determinism introduction and elimination*
Intent	use different algorithms for the same problem
Applicability	some algorithm is too slow or it may be incorrect
Structure	either replace non-deterministic code with deterministic one or vice versa
Consequences	improve speed or verify correctness of algorithms
Known uses	prototyping
See also	

Specifications of problems are often non-deterministic because in many cases non-determinism defines the desired results of a computation more easily than by other means. We have seen this practice in several previous examples. Functional logic programming, more than any other paradigm, allows a programmer to translate a specification, whether or not non-deterministic, with little or no change into a program [1]. Thus, it is not unusual for programmers to initially code non-deterministic programs even for deterministic problems because this approach produces correct programs quickly. We call a *prototypical implementation* this direct encoding of a specification.

For some problems, prototypical implementations are not as efficient as an application requires. This is typical, e.g., for sorting and searching problems, which have been the subject of long investigations, because non-deterministic solutions ignore domain knowledge that speeds up computations. In these cases, the prototypical implementation, often non-deterministic, should be replaced by a more efficient implementation, often deterministic, that produces the same result. We call the latter *production implementation*. The investment that went into the prototypical implementation is not wasted, as several benefits derive from that effort. First of all, the specification of the problem is better understood and it has been tested through the input/output behavior of the prototypical implementation and possibly debugged and corrected. Second, the prototypical implementation can be used as a testing oracle of the production implementation. Testing can be largely automated, which both reduces effort and improves reliability.

PAKCS [20] is distributed with a unit testing tool [13] called *CurryTest* which is useful in the situation we describe. A unit testing of a program is another program defining, among others, some zero-arity functions containing assertions, e.g., specific conditions stating the equality between a function call and its result. The *CurryTest* tool applied to the program invokes, using reflections, all the functions defining an assertion. The tool checks the validity of each assertion and reports any violation. Thus, a test of a function f of the production imple-

mentation will apply both f and the corresponding function of the prototypical implementation to test arguments, and assert that the results are the same.

Example: Assume an informal specification of sorting: "find the minimum of a list, sort the rest, and place the minimum at the front." The most complicated aspect of an implementation of this specification is the computation of the minimum. A fairly precise specification of the minimum was given in the *Quantification* pattern. A prototypical implementation of this specification follows:

```
getMinSpec :: [a] -> a
getMinSpec l | exists m (\m -> m =:= elemOf l &>
                     notExists (elemOf'set l) (\x -> x < m))
             = m  where m free
```

The prototypical implementation is assumed to be correct, since it is a direct encoding of the specification, and it is non-deterministic due to `elemOf`. Sorting with the prototypical implementation is too slow for long lists, hence we code a deterministic and more efficient production implementation:

```
getMin :: [a] -> a
getMin (x:xs) = aux x xs
              where aux x [] = x
                    aux x (y:ys) | x <= y = aux x ys
                                 | otherwise = getMin (y:ys)
```

To test the production implementation, we define the following function that compares the output of the production implementation with that of the prototypical implementation. We process this function with *CurryTest*.

```
testGetMin :: Assertion Int
testGetMin = AssertEqual "getMin" outSpec out
           where input = [3,1,2,4,9,5] -- some test data
                 outSpec = getMinSpec input
                 out     = getMin     input
```

This pattern is typically used to replace non-deterministic code with more deterministic code to improve the efficiency of a program. However, the opposite replacement is occasionally useful to attempt to "improve the correctness" of a program. Suppose that the input/output behavior of a program is incorrect and that we suspect that the culprit is some function f. We can replace the code of f with code directly obtained from the specification of f. This code is likely to be more non-deterministic. If the replacement fixes the input/output behavior of the program, we have the proof that the code that we replaced was indeed incorrect.

Both unit testing and replacing an algorithm with another for various purposes, such as improving performance or verifying behavior, are widespread and language independent techniques. The customization of these techniques to functional logic programming emphasizes the possibility of executing non-deterministic code, in particular code obtained from a direct encoding of a specification.

4 Conclusion and Related Work

Design patterns help structuring code for general problems frequently arising in software development. They produce solutions that are more readable, maintainable and elegant than improvised alternatives. Efficiency may be partially sacrificed for these very desirable attributes. We have also shown that employing a pattern has various benefits even in situations in which the pattern-driven solution is inefficient and it is eventually replaced by more efficient code.

We presented five new design patterns for the functional logic programming paradigm. These patterns are a follow up on our initial work [3] in this area. Patterns distill successful programming experience similar in scope to programming pearls. Some of our patterns were motivated, in part, by features introduced in the Curry language in the last 10 years, in particular functional patterns [4] and set functions [5].

The programs discussed in this paper are available at URL:

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

References

1. S. Antoy. Programming with narrowing. *Journal of Symbolic Computation*, 45(5):501–522, May 2010.
2. S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *Journal of the ACM*, 47(4):776–822, 2000.
3. S. Antoy and M. Hanus. Functional logic design patterns. In *6th Int'l Symp, on Functional and Logic Programming (FLOPS'02)*, pages 67–87, Aizu, Japan, 9 2002. Springer LNCS 2441.
4. S. Antoy and M. Hanus. Declarative programming with function patterns. In *15th Int'nl Symp. on Logic-based Program Synthesis and Transformation (LOPSTR 2005)*, pages 6–22, London, UK, Sept. 2005. Springer LNCS 3901.
5. S. Antoy and M. Hanus. Set functions for functional logic programming. In *Proceedings of the 11th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP 2009)*, pages 73–82, Lisbon, Portugal, September 2009.
6. K. Beck and W. Cunningham. Using pattern languages for object-oriented programs. In *Specification and Design for Object-Oriented Programming (OOPSLA-87)*, 1987.
7. 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.
8. R. Caballero and J. Sánchez, editors. *TOY: A Multiparadigm Declarative Language (version 2.3.1)*, 2007. Available at `http://toy.sourceforge.net`.
9. Rafael Caballero and Francisco Javier López-Fraguas. Extensions: A technique for structuring functional-logic programs. In *Proceedings of the Third International Andrei Ershov Memorial Conference on Perspectives of System Informatics*, PSI '99, pages 297–310, London, UK, 2000. Springer-Verlag.
10. J. W. Cooper. *Java Design Patterns*. Addison Wesley, 2000.
11. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

12. M. Grand. *Patterns in Java*. J. Wiley, 1998.
13. M. Hanus. Currytest: A tool for testing Curry programs. Available at `http://www-ps.informatik.uni-kiel.de/currywiki/tools/currytest`. Accessed April 13, 2011.
14. 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.
15. 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.
16. 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.
17. 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.
18. M. Hanus. Type-oriented construction of web user interfaces. In *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'06)*, pages 27–38. ACM Press, 2006.
19. M. Hanus. Declarative processing of semistructured web data. Technical report 1103, Christian-Albrechts-Universität Kiel, 2011.
20. M. Hanus, S. Antoy, B. Braßel, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, and F. Steiner. PAKCS: The Portland Aachen Kiel Curry System. Available at `http://www.informatik.uni-kiel.de/~pakcs/`, 2011.
21. 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.
22. M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.8.2). Available at `http://www.informatik.uni-kiel.de/~curry`, March 28, 2006.
23. Paul Hudak, John Peterson, and Joseph Fasel. A gentle introduction to Haskell 98. Available at `http://www.haskell.org/tutorial/monads.html`, 1999.
24. F. J. López-Fraguas and J. Sánchez-Hernández. TOY: A multiparadigm declarative system. In *Proceedings of the Tenth International Conference on Rewriting Techniques and Applications (RTA'99)*, pages 244–247. Springer LNCS 1631, 1999.
25. M. J. O'Donnell. *Equational Logic as a Programming Language*. MIT Press, 1985.
26. S.L. Peyton Jones and J. Hughes. Haskell 98: A non-strict, purely functional language. `http://www.haskell.org`, 1999.
27. V.A. Saraswat. *Concurrent Constraint Programming*. MIT Press, 1993.
28. Wikipedia, the free encyclopedia. Four color theorem. Available at `http://en.wikipedia.org/wiki/Four_color_theorem`. Accessed April 8, 2011.