Dynamic Predicates in Functional Logic Programs*

Michael Hanus†

October 15, 2004

Abstract

In this paper we propose a new concept to deal with dynamic predicates in functional logic programs. The definition of a dynamic predicate can change over time, i.e., one can add or remove facts that define this predicate. Our approach is easy to use and has a clear semantics that does not depend on the particular (demand-driven) evaluation strategy of the underlying implementation. In particular, the concept is not based on (unsafe) side effects so that the order of evaluation does not influence the computed results—an essential requirement in non-strict languages.

Dynamic predicates can also be persistent so that their definitions are saved across invocations of programs. Thus, dynamic predicates are a lightweight alternative to the explicit use of external database systems. Moreover, they extend one of the classical application areas of logic programming to functional logic programs. We present the concept, its use and an implementation in a Prolog-based compiler.

1 Motivation and Related Work

Functional logic languages [11] aim to integrate the best features of functional and logic languages in order to provide a variety of programming concepts to the programmer. For instance, the concepts of demand-driven evaluation, higher-order functions, and polymorphic typing from functional programming can be combined with logic programming features like computing with partial information (logical variables), constraint solving, and non-deterministic search for solutions. This combination leads to optimal evaluation strategies [2] and new design patterns [4] that can be applied to provide better programming abstractions, e.g., for implementing graphical user interfaces [13] or programming dynamic web pages [14].

However, one of the traditional application areas of logic programming is not yet sufficiently covered in existing functional logic languages: the combination

*This research has been partially supported by the German Research Council (DFG) under grants Ha 2457/1-2 and Ha 2457/5-1.

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

of declarative programs with persistent information, usually stored in relational databases, that can change over time. Logic programming provides a natural framework for this combination (e.g., see [8, 10]) since externally stored relations can be considered as facts defining a predicate of a logic program. Thus, logic programming is an appropriate approach to deal with deductive databases or declarative knowledge management.

In this paper, we propose a similar concept for functional logic languages. Nevertheless, this is not just an adaptation of existing concepts to functional logic programming. We will show that the addition of advanced functional programming concepts, like the clean separation of imperative and declarative computations by the use of monads [27], provides a better handling of the dynamic behavior of database predicates, i.e., when we change the definition of such predicates by adding or removing facts. To motivate our approach, we shortly discuss the problems caused by traditional logic programming approaches to dynamic predicates.

The logic programming language Prolog allows to change the definition of predicates[1] by adding or deleting clauses using predefined predicates like `asserta` (adding a new *first* clause), `assertz` (adding a new *last* clause), or `retract` (deleting a matching clause). Problems occur if the use of these predicates is mixed with their update. For instance, if a new clause is added during the evaluation of a literal, it is not directly clear whether this new clause should be visible during backtracking, i.e., a new proof attempt for the same literal. This has been discussed in [21] where the so-called "logical view" of database updates is proposed. In the logical view, only the clauses that exist at the first proof attempt to a literal are used. Although this solves the problems related to backtracking, advanced evaluation strategies cause new problems.

It is well known that advanced control rules, like coroutining, provide a better control behavior w.r.t. the termination and efficiency of logic programs [24]. Although the completeness of SLD resolution w.r.t. any selection rule seems to justify such advanced control rules, it is not the case w.r.t. dynamic predicates. For instance, consider the Prolog program

```
ap(X) :- assertz(p(X)).

q :- ap(X), p(Y), X=1.
```

If there are no clauses for the dynamic predicate `p`, the proof of the literal `q` succeeds due to the left-to-right evaluation of the body of the clause for `q`. However, if we add the block declaration (in Sicstus-Prolog) "`:- block ap(-).`" to specify that `ap` should be executed only if its argument is not a free variable, then the proof of the literal `q` fails, because the clause for `p` has not been asserted when `p(Y)` should be proved.

This example indicates that care is needed when combining dynamic predicates and advanced control strategies. This is even more important in functional logic languages that are usually based on demand-driven (and concurrent) eval-

[1] In many Prolog systems, such predicates must be declared as "dynamic" in order to change their definitions dynamically.

uation strategies where the exact order of evaluation is difficult to determine in advance [2, 12].

Unfortunately, existing approaches to deal with dynamic predicates do not help here. For instance, Prolog and its extensions to persistent predicates stored in databases, like the Berkeley DB of Sicstus-Prolog or the persistence module of Ciao Prolog [7], suffer from the same problems. In the other hand, functional language bindings to databases do not offer the constraint solving and search facilities of logic languages. For instance, HaSQL[2] supports a simple connection to relational databases via I/O actions but provides no abstraction for computing queries (the programmer has to write SQL queries in plain text). This is improved in Haskell/DB [6, 20] which allows to express queries through the use of specific operators. More complex information must be deduced by defining appropriate functions.

Other approaches to integrate functional logic programs with databases concentrate only on the semantical model for query languages. For instance, [1] proposes an integration of functional logic programming and relational databases by an extended data model and relational calculus. However, the problem of database updates is not considered and an implementation is not provided. Echahed and Serwe [9] propose a general framework for functional logic programming with processes and updates on clauses. Since they allow updates on arbitrary program clauses (rather than facts), it is unclear how to achieve an efficient implementation of this general model. Moreover, persistence is not covered in their approach.

Since real applications require the access and manipulation of persistent data, we propose a new model to deal with dynamic predicates in functional logic programs where we choose the declarative multi-paradigm language Curry [18] for concrete examples.[3] Although the basic idea is motivated by existing approaches (a dynamic predicate is considered as defined by a set of basic facts that can be externally stored), we propose a clear distinction between the accesses and updates to a dynamic predicate. In order to abstract from the concrete (demand-driven) evaluation strategy, we propose the use of time stamps to mark the lifetime of individual facts.

Dynamic predicates can also be persistent so that their definitions are saved across invocations of programs. Thus, our approach to dynamic predicates is a lightweight alternative to the explicit use of external database systems that can be easily applied. Nevertheless, one can also store dynamic predicates in an external database if the size of the dynamic predicate definitions becomes too large.

The next section provides a basic introduction into Curry. Section 3 contains a description of our proposal to integrate dynamic predicates into functional logic languages. Section 4 sketches a concrete implementation of this concept. Section 5 discusses extensions of the basic concept and Section 6 contains our

[2] `http://members.tripod.com/~sproot/hasql.htm`

[3] Our proposal can be adapted to other modern functional logic languages that are based on the monadic I/O concept to integrate imperative and declarative computations in a clean manner, like Escher [22], Mercury [26], or Toy [23].

conclusions.

2 Curry in a Nutshell

In this section we review those elements of Curry which are necessary to understand the contents of this paper. More details about Curry's computation model and a complete description of all language features can be found in [12, 18].

Curry is a modern multi-paradigm declarative language combining in a seamless way features from functional, logic, and concurrent programming and supporting programming-in-the-large with specific features (types, modules, encapsulated search). From a syntactic point of view, a Curry program is a functional program extended by the possible inclusion of free (logical) variables in conditions and right-hand sides of defining rules. Curry has a Haskell-like syntax [25], 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 ("$f\ e$").

A Curry *program* consists of the definition of functions and data types on which the functions operate. Functions are evaluated lazily. To provide the full power of logic programming, functions can be called with partially instantiated arguments and defined by conditional equations with constraints in the conditions. The behavior of function calls with free variables depends on the evaluation mode of functions which can be either *flexible* or *rigid*. Calls to flexible functions are evaluated by a possibly non-deterministic instantiation of the demanded arguments (i.e., arguments whose values are necessary to decide the applicability of a rule) to the required values in order to apply a rule ("*narrowing*"). Calls to rigid functions are suspended if a demanded argument is uninstantiated ("*residuation*").

Example 2.1 The following Curry program defines the data types of Boolean values, "possible" (maybe) values, and polymorphic lists (first three lines) and functions for computing the concatenation of lists and the last element of a list:

```
data Bool    = True    | False
data Maybe a = Nothing | Just a
data List a  = []      | a : List a

conc :: [a] -> [a] -> [a]
conc []     ys = ys
conc (x:xs) ys = x : conc xs ys

last :: [a] -> a
last xs | conc ys [x] =:= xs   = x   where x,ys free
```

The data type declarations define `True` and `False` as the Boolean constants, `Nothing` and `Just` as the constructors for possible values (where `Nothing` is considered as no value), and `[]` (empty list) and `:` (non-empty list) as the constructors for polymorphic lists (`a` is a type variable ranging over all types

and the type "`List a`" is usually written as `[a]` for conformity with Haskell).

The (optional) type declaration ("`::`") of the function `conc` specifies that `conc` takes two lists as input and produces an output list, where all list elements are of the same (unspecified) type.[4] Since `conc` is flexible,[5] the equation "`conc ys [x] =:= xs`" is solved by instantiating the first argument `ys` to the list `xs` without the last argument, i.e., the only solution to this equation satisfies that `x` is the last element of `xs`.

In general, functions are defined by (*conditional*) *rules* of the form

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

with f being a function, $t_1, \ldots, t_n$ *patterns* (i.e., expressions without defined functions) without multiple occurrences of a variable, the *condition* c is a constraint, e is a well-formed *expression* which may also contain function calls, lambda abstractions etc, and vs is the list of *free variables* that occur in c and e but not in $t_1, \ldots, t_n$. The condition and the `where` parts can be omitted if c and vs are empty, respectively. The `where` part can also contain further local function definitions which are only visible in this rule. A conditional rule can be applied if its left-hand side matches the current call and its condition is satisfiable.

A *constraint* is any expression of the built-in type `Success`. For instance, the trivial constraint `success` is an expression of type `Success` that denotes the always satisfiable constraint. "c_1 `&` c_2" denotes the *concurrent conjunction* of the constraints c_1 and c_2, i.e., this expression is evaluated by proving both argument constraints concurrently. Each Curry system provides at least *equational constraints* of the form e_1 `=:=` e_2 which are satisfiable if both sides e_1 and e_2 are reducible to unifiable patterns. However, specific Curry systems can also support more powerful constraint structures, like arithmetic constraints on real numbers or finite domain constraints, as in the PAKCS implementation [15].

Predicates in the sense of logic programming can be considered as functions with result type `Success`. For instance, a predicate `isPrime` that is satisfied if the argument (an integer number) is a prime can be modeled as a function with type

```
isPrime :: Int -> Success
```

The following rules define a few facts for this predicate:

```
isPrime 2 = success
isPrime 3 = success
isPrime 5 = success
isPrime 7 = success
```

Apart from syntactic differences (that support, in contrast to pure logic programming, the use of predicates and partial applications of predicates as first-

[4] Curry uses curried function types where α`->`β denotes the type of all functions mapping elements of type α into elements of type β.

[5] As a default, all functions except for I/O actions and external functions are flexible.

class citizens in higher-order functions), any pure logic program has a direct correspondence to a Curry program. For instance, a predicate `isPrimePair` that is satisfied if the arguments are primes that differ by 2 can be defined as follows:

```
isPrimePair :: Int -> Int -> Success
isPrimePair x y = isPrime x & isPrime y & x+2 =:= y
```

The operational semantics of Curry, precisely described in [12, 18], is based on an optimal evaluation strategy [2] which is a conservative extension of lazy functional programming and (concurrent) logic programming. Due to its demand-driven behavior, it provides optimal evaluation (e.g., shortest derivation sequences, minimal solution sets) on well-defined classes of programs (see [2] for details). Curry also offers the standard features of functional languages, like higher-order functions (e.g., "`\`x `->` e" denotes an anonymous function that assigns to each x the value of e) or monadic I/O. Since the latter is important for the ideas in this paper, we sketch the I/O concept of Curry which is almost identical to the monadic I/O of Haskell [27].

In the monadic approach to I/O, an interactive program is considered as a function computing a sequence of actions that are applied to the outside world. An *action* changes the state of the world and possibly returns a result (e.g., a character read from the terminal). Thus, actions are functions of type

World `->` `(`α`,`*World*`)`

(where *World* denotes the type of all states of the outside world). This function type is also abbreviated by `IO` α. If an action of type `IO` α is applied to a particular world, it yields a value of type α and a new (changed) world. For instance, `getChar` of type `IO Char` is an action which reads a character from the standard input whenever it is executed, i.e., applied to a world. The important point is that values of type *World* are not accessible to the programmer—she/he can only create and compose actions on the world. For instance, the action `getChar` can be composed with the action `putChar` (which has type `Char -> IO ()` and writes a character to the terminal) by the sequential composition operator `>>=` (which has type `IO` α `->` `(`α `->` `IO` β`)` `->` `IO` β), i.e., "`getChar >>= putChar`" is a composed action that prints the next character of the input stream on the screen. The second composition operator `>>` is like `>>=` but ignores the result of the first action. Furthermore, `done` is the "empty" action which does nothing (see [27] for more details). For instance, a function which takes a string (list of characters) and produces an action that prints it to the terminal followed by a line feed can be defined as follows:

```
putStrLn []     = putChar '\n'
putStrLn (c:cs) = putChar c >> putStrLn cs
```

It should be noted that an action is executed when the program (applied to the world) is executed. Since the world cannot be copied, non-deterministic actions as a result of a program are not allowed. Therefore, all possible search must

be encapsulated between I/O operations using the features for encapsulating search [5, 17].

3 Dynamic Predicates

In this section we describe our proposal to dynamic predicates in functional logic programs and show its use by several examples.

3.1 General Concept

Since the definition of dynamic predicates is also intended to be stored persistently in files, we assume that dynamic predicates are defined by ground (i.e., variable-free) facts.[6] However, in contrast to predicates that are explicitly defined in a program (e.g., `isPrime` in Section 2), the definition of a *dynamic* predicate is not provided in the program code but will be dynamically computed. Thus, dynamic predicates are similar to "external" functions whose code is not contained in the program but defined elsewhere. Therefore, the programmer has to specify in a program only the (monomorphic) type signature of a dynamic predicate (remember that Curry is strongly typed) and mark its name as "`dynamic`".

As a simple example, we want to define a dynamic predicate `prime` to store prime numbers whenever we compute them. Thus, we provide the following definition in our program:

```
prime :: Int -> Dynamic
prime dynamic
```

Similarly to `Success`, the predefined type "`Dynamic`" is abstract, i.e., there are no accessible data constructors of this type but a few predefined operations that act on objects of this type (see below). From a declarative point of view, `Dynamic` is similar to `Success`, i.e., `prime` can be considered as a predicate. However, since the definition of dynamic predicates may change over time, the access to dynamic predicates is restricted in order to avoid the problems mentioned in Section 1. Thus, the use of the type `Dynamic` ensures that the specific access and update operations (see below) can be applied only to dynamic predicates. Furthermore, the keyword "`dynamic`" informs the compiler that the code for `prime` is not in the program but externally stored (similarly to the definition of external functions).

In order to avoid the problems related to mixing update and access to dynamic predicates, we put the corresponding operations into the I/O monad since this ensures a sequential evaluation order. Thus, we provide the following predefined operations:

[6] If one wants to store non-ground facts or rules, e.g., in applications related to knowledge management, one can also use a ground representation together with appropriate meta-programming techniques. Thus, our requirement to ground facts is not a real restriction in practice.

```
assert :: Dynamic -> IO ()

retract :: Dynamic -> IO Bool

getKnowledge :: IO (Dynamic -> Success)
```

`assert` adds a new fact about a dynamic predicate to the database where the *database* is considered as the set of all known facts for dynamic predicates. Actually, the database can also contain multiple entries (if the same fact is repeatedly asserted) so that the database is a multi-set of facts. For the sake of simplicity, we ignore this detail and talk about sets in the following.

Since the facts defining dynamic predicates do not contain unbound variables (see above), `assert` is a rigid function, i.e., it suspends when the arguments (after evaluation to normal form) contain unbound variables. Similarly, `retract` is also rigid[7] and removes a matching fact, if possible (this is indicated by the Boolean result value). For instance, the sequence of actions

```
assert (prime 1) >> assert (prime 2) >> retract (prime 1)
```

asserts the new fact `(prime 2)` to the database.

The action `getKnowledge` is intended to retrieve the set of facts stored in the database at the time when this action is executed. In order to provide access to the set of facts, `getKnowledge` returns a function of type "`Dynamic -> Success`" which can be applied to expressions of type "`Dynamic`", i.e., calls to dynamic predicates. For instance, the following sequence of actions asserts a new fact `(prime 2)` and retrieves its contents by unifying the logical variable `x` with the value 2:[8]

```
assert (prime 2) >> getKnowledge >>= \known ->
                    doSolve (known (prime x))
```

Since writing monadic sequences of I/O actions is not well readable, we use Haskell's "`do`" notation [25]. Thus, we write the previous action sequence in the following form:

```
do assert (prime 2)
   known <- getKnowledge
   doSolve (known (prime x))
```

Since there might be several facts that match a call to a dynamic predicate, we have to encapsulate the possible non-determinism occurring in a logic computation. This can be done in Curry by the primitive action to encapsulate the search for all solutions to a goal:

```
getAllSolutions :: (a -> Success) -> IO [a]
```

[7] One could argue that `retract` could be also called with logical variables which should be bound to the values of the retracted facts; however, this might cause non-deterministic actions (if more than one fact matches) which leads to run-time errors.

[8] The action `doSolve` is defined as "`doSolve c | c = done`" and can be used to embed constraint solving into the I/O monad.

`getAllSolutions` takes a constraint abstraction and returns the list of all solutions, i.e., all values for the argument of the abstraction such that the constraint is satisfiable.[9] For instance, the evaluation of

```
getAllSolutions (\x -> known (prime x))
```

returns the list of all values for `x` such that `known (prime x)` is satisfied. Thus, we can define a function `printKnownPrimes` that prints the list of all known prime numbers as follows:

```
printKnownPrimes = do
   known <- getKnowledge
   sols <- getAllSolutions (\x -> known (prime x))
   print sols
```

If we just want to check whether a particular fact of a dynamic predicate is known, we can define the following function:

```
isKnown :: Dynamic -> IO Bool
isKnown p = do
   known <- getKnowledge
   sols <- getAllSolutions (\_ -> known p)
   return (sols /= [])
```

Here we are not interested in individual solutions. Thus, we write the anonymous variable "`_`" as the argument to the search goal and finally check whether some solution has been computed.

Note that we can use all logic programming techniques also for dynamic predicates: we just have to pass the result of `getKnowledge` (i.e., the variable `known` above) into the clauses defining the deductive part of the database program and wrap all calls to a dynamic predicate with this result variable. For instance, we can print all prime pairs by the following definitions:

```
primePair known (x,y) =
   known (prime x) & known (prime y) & x+2 =:= y

printPrimePairs = do
   known <- getKnowledge
   sols <- getAllSolutions (\p -> primePair known p)
   print sols
```

The constraint `primePair` specifies the property of being a prime pair w.r.t. the knowledge `known`, and the action `printPrimePairs` prints all currently known prime pairs.

If one wants to avoid passing the variable `known` through all predicates that do inferences on the current knowledge, one can also define these predicates locally so that `known` becomes automatically visible to all predicates. In order

[9] `getAllSolutions` is an I/O action since the order of the result list might vary from time to time due to the order of non-deterministic evaluations.

to write the program code even more in the logic programming style, we define the composition of `known` and `prime` as a single name. The following code, which defines a constraint for non-empty sequences of ascending primes, shows an example for this "LP" style ("`.`" denotes function composition):

```
primeSequence known l = primes l

 where

  isPrime = known . prime

  primes [p] = isPrime p
  primes (p1:p2:ps) = isPrime p1 &
                      isPrime p2 &
                      (p1<p2) =:= True &
                      primes (p2:ps)
```

Our concept provides a clean separation between database updates and accesses. Since we get the knowledge at a particular point of time, we can access all facts independent on the order of evaluation. Actually, the order is difficult to determine due to the demand-driven evaluation strategy. For instance, consider the following sequence of actions:

```
do assert (prime 2)
   known1 <- getKnowledge
   assert (prime 3)
   assert (prime 5)
   known2 <- getKnowledge
   sols2 <- getAllSolutions (\x -> known2 (prime x))
   sols1 <- getAllSolutions (\x -> known1 (prime x))
   return (sols1,sols2)
```

Executing this code with the empty database, the pair of lists `([2],[2,3,5])` is returned. Although the concrete computation of all solutions is performed later than they are conceptually accessed (by `getKnowledge`) in the program text, we get the right facts (in contrast to Prolog with coroutining, see Section 1). Therefore, `getKnowledge` conceptually copies the current database for later access. However, since an actual copy of the database can be quite large, this is implemented by the use of time stamps (see Section 4).

3.2 Persistent Dynamic Predicates

One of the key features of our proposal is the easy handling of persistent data. The facts about dynamic predicates are usually stored in main memory which supports fast access. However, in most applications it is necessary to store the data also persistently so that the actual definitions of dynamic predicates survive different executions (or crashes) of the program. One approach is to store the facts in relational databases (which is non-trivial since we allow arbitrary term structures as arguments). Another alternative is to store them in files (e.g., in

XML format). In both cases the programmer has to consider the right format and access routines for each application. Our approach is much simpler (and often also more efficient if the size of the dynamic data is not extremely large): it is only necessary to declare the predicate as "persistent". For instance, if we want to store our knowledge about primes persistently, we define the predicate `prime` as follows:

```
prime :: Int -> Dynamic
prime persistent "file:prime_infos"
```

Here, `prime_infos` is the name of a directory where the run-time system automatically puts all files containing information about the dynamic predicate `prime`.[10] Apart from changing the `dynamic` declaration into a `persistent` declaration, nothing else needs to be changed in our program. Thus, the same actions like `assert`, `retract`, or `getKnowledge` can be used to change or access the persistent facts of `prime`. Nevertheless, the persistent declaration has important consequences:

- All facts and their changes are persistently stored, i.e., after a termination (or crash) and restart of the program, all facts are automatically recovered.
- Changes to dynamic predicates are immediately written into a log file so that they can be recovered.
- `getKnowledge` gets always the current knowledge persistently stored, i.e., if other processes also change the facts of the same predicate, it becomes immediately visible with the next call to `getKnowledge`.
- In order to avoid conflicts between concurrent processes working on the same dynamic predicates, there is also a transaction concept (see Section 3.3).

Note that the easy and clean addition of persistency was made possible due to our concept to separate the update and access to dynamic predicates. Since updates are put into the I/O monad, there are obvious points where changes must be logged. On the other hand, the `getKnowledge` action needs only a (usually short) synchronization with the external data and then the knowledge can be used with the efficiency of the internal program execution.

3.3 Transactions

The persistent storage of dynamic predicates causes another problem: if several concurrent processes updates the same data, some synchronization is necessary. Since we intend to use our proposal also for web applications [14], there is a clear need to solve the synchronization problem since in such applications one

[10] The prefix "`file:`" instructs the compiler to use a file-based implementation of persistent predicates. For future work, it is planned also to use relational databases to store persistent facts so that this prefix is used to distinguish the different access methods.

does not know when the individual programs reacting to client's requests are executed. Fortunately, the database community has solved this problem via transaction models so that we only have to adapt them into our framework of functional logic programming.

We consider a *transaction* as a sequence of changes to (possibly several different) dynamic predicates that should only be performed together or completely ignored. Moreover, the changes of a transaction become visible to other concurrent processes only if the complete transaction has been successfully executed. This model can be easily supported by providing two I/O actions:

```
transaction :: IO a -> IO (Maybe a)

abortTransaction :: IO a
```

`transaction` takes an I/O action (usually, a sequence of updates to dynamic predicates) as argument and tries to execute it. If this was successfully done, the result r of the argument action is returned as `(Just` r`)` and all changes to the dynamic predicates become visible to other processes. Otherwise, i.e., in case of a failure, run-time error, or if the action `abortTransaction` has been executed, all changes to dynamic predicates performed during this transaction are undone and `Nothing` is returned to indicate the failure of the transaction. For instance, consider the following transaction:

```
try42 = do assert (prime 42)
           abortTransaction
           assert (prime 43)
```

If we execute "`transaction try42`", then no change to the definition of the persistent dynamic predicate `prime` becomes visible.

4 Implementation

In order to test our concept and to provide a reasonable implementation, we have implemented it in the PAKCS implementation of Curry [15]. This implementation is fairly efficient and has been used for many non-trivial applications, e.g., a web-based system for e-learning [16]. The system compiles Curry programs into Prolog by transforming pattern matching into predicates and exploiting coroutining for the implementation of the concurrency features of Curry [3]. Due to the use of Prolog as the back-end language, the implementation of our concept is not very difficult. Therefore, we highlight only a few aspects of this implementation.

First of all, the compiler of PAKCS has to be adapted since the code for dynamic predicates must be different from other functions. Thus, the compiler translates a declaration of a dynamic predicate into specific code so that the run-time evaluation of a call to a dynamic predicate yields a data structure containing information about the actual arguments and the name of the external database (in case of persistent predicates). In this implementation, we have not

used a relational database for storing the facts since this is not necessary for the size of the dynamic data (in our applications only a few megabytes). Instead, all facts are stored in main memory and in files in case of persistent predicates. First, we describe the implementation of non-persistent predicates.

Each `assert` and `retract` action is implemented via Prolog's `assert` and `retract`. However, as additional arguments we use time stamps to store the lifetime (birth and death) of all facts in order to implement the visibility of facts for the `getKnowledge` action (similarly to [21]). Thus, there is a global clock ("update counter") in the program that is incremented for each `assert` and `retract`. If a fact is asserted, it gets the actual time as birth time and ∞ as the death time. If a fact is retracted, it is not retracted in memory but only the death time is set to the actual time since there might be some unevaluated expression for which this fact is still visible. `getKnowledge` is implemented by returning a predefined function that keeps the current time as an argument. If this function is applied to some dynamic predicate, it unifies the predicate with all facts and, in case of a successful unification, it checks whether the time of the `getKnowledge` call is in the birth/death interval of this fact.

Persistent predicates are similarly implemented, i.e., all known facts are always kept in main memory. However, each update to a persistent predicate is written into a log file. Furthermore, all facts of this predicate are stored in a file in Prolog format. This file is only read and updated in the first call to `getKnowledge` or in subsequent calls if another concurrent process has changed the persistent data. In this case, the following operations are performed:

1. The previous database file with all Prolog facts is read.
2. All changes from the log file are replayed, i.e., executed.
3. A new version of the database file is written.
4. The log file is cleared.

In order to avoid problems in case of program crashes during this critical period, the initialization phase is made exclusive to one process via operating system locks and backup files are written.

To reduce the time to load the database, we store it also in an intermediate format (Prolog object file format of Sicstus-Prolog). With this binary format, the database for most applications can be loaded very efficiently. For instance, it needs 120 milliseconds to load a database of 12.5 MB Prolog source code on a 2.0 GHz Linux-PC (AMD Athlon XP 2600 with 256 KB cache).

In order to implement transactions and the concurrent access to persistent data, operating system locks are used. Moreover, version numbers of the database are stored in order to inform the running program about changes to the database by other processes. These changes are taken into account when `getKnowledge` is executed. Transactions are implemented by writing marks into the log files and considering only complete transactions when recovering the database in the initialization phase described above.

5 Extending the Basic Interface

The concept to deal with dynamic predicates described in Section 3 is easy to use and sufficiently expressive as shown by various examples. Nevertheless, we propose in this section a few combinators that simplify the construction of more complex queries related to dynamic predicates.

Non-trivial queries often involve more than one dynamic predicate (or table in relational databases). Thus, it is quite useful to provide a combinator to join two dynamic predicates:

```
(<>) :: Dynamic -> Dynamic -> Dynamic
```

Using this combinator, we can write "`prime x <> prime y`" to access two prime numbers `x` and `y`. Furthermore, dynamic predicates are usually restricted with a Boolean condition so that the following combinator is useful:

```
(|>) :: Dynamic -> Bool -> Dynamic
```

If the operators are defined so that "`<>`" binds stronger then "`|>`", then the expression "`prime x <> prime y |> x+2==y`" specifies numbers `x` and `y` that are prime pairs.

The implementation of these combinators does not require a real extension of the basic concept described in Section 3, since it can be implemented on top of the primitives introduced so far. For this purpose, we introduce the following data type to specify complex queries on dynamic predicates:

```
data Dynamic = Pred PrimDynamic
             | Prod Dynamic Dynamic
             | Cond Dynamic Bool
```

`PrimDynamic` denotes the type of dynamic predicates as introduced in Section 3, i.e., we assume that an expression like "`prime x`" returns a value wrapped with the constructor `Pred`. Then we define "`<>`" and "`|>`" as renamings of the corresponding constructors:

```
d1 <> d2 = Prod d1 d2

d  |> b  = Cond d b
```

Next, we extend the base operations `assert`, `retract`, and `getKnowledge` of Section 3 on this data type (where we prefix the original operations of Section 3 by "`prim_`"). For instance, `assert` adds new facts for all predicates in the dynamic expression provided that the condition holds in case of conditional dynamics:

```
assert :: Dynamic -> IO ()
assert (Pred pred)  = prim_assert pred
assert (Prod d1 d2) = assert d1 >> assert d2
assert (Cond d b)   = if b then assert d
                           else done
```

Thus, we can assert several facts using the combinator `<>`, as in

```
assert (prime 2 <> prime 3 <> prime 5 <> prime 7 <> prime 11)
```

Exploiting the higher-order programming features of Curry, we can write the previous expression also in the form

```
assert (foldr1 (<>) (map prime [2,3,5,7,11]))
```

which could be useful for asserting larger data sets.

Similarly, `retract` deletes all facts and returns `True` if all retracted facts exist:

```
retract :: Dynamic -> IO Bool
retract (Pred pred)  = prim_retract pred
retract (Prod d1 d2) = do b1 <- retract d1
                          b2 <- retract d2
                          return (b1 && b2)
retract (Cond d b)   = if b then retract d
                            else return True
```

Analogously, we define the extension of `getKnowledge` to complex dynamic expressions:

```
getKnowledge :: IO (Dynamic -> Success)
getKnowledge = do known <- prim_getKnowledge
                  return (knownAll known)
 where
   knownAll k (Pred pred)  = k pred
   knownAll k (Prod d1 d2) = knownAll k d1 & knownAll k d2
   knownAll k (Cond d b)   = knownAll k d & b=:=True
```

Using these definitions, we can use the basic operations on dynamic predicates identical to Section 3 but support also the processing of complex queries constructed with the operators `<>` and `|>`.

As we have seen in Section 3, the typical access to dynamic data consists of retrieving the current contents of the database (`getKnowledge`) and accessing the individual data by computing all solutions to a constraint involving dynamic predicates wrapped with a call to `getAllSolutions`. We can define this standard combination of `getKnowledge` and `getAllSolutions` as a single function:

```
getDynamicSolutions :: (a -> Dynamic) -> IO [a]
getDynamicSolutions query = do
  known <- getKnowledge
  getAllSolutions (\x -> known (query x))
```

Thus, `getDynamicSolutions` takes an abstraction on a dynamic expression and returns all solutions to this abstraction. For instance, we can rewrite the action to print all prime pairs (see Section 3.1) in the following form:

```
printPrimePairs = do
  sols <- getDynamicSolutions
                    (\(x,y) -> prime x <> prime y |> x+2==y)
  print sols
```

Note that the argument to `getDynamicSolutions` has all the elements of a typical SQL query (which is of the general form `SELECT ... FROM ... WHERE ...`): the argument "`(x,y)`" is the projection on the attributes of interest (`SELECT`), "`prime x <> prime y`" refers to the involved relations (`FROM`), and the condition "`x+2==y`" restricts the number of potential values (`WHERE`).

Due to the embedding of dynamic predicates in a functional logic programming language, we can also formulate recursive queries. For instance, non-empty ascending sequences of primes can be printed as follows (compare function `primeSequence` in Section 3.1):

```
printPrimeSequences = do
  sols <- getDynamicSolutions primes
  print sols
 where
  primes [p] = prime p
  primes (p1:p2:ps) = (prime p1 <> prime p2 |> p1<p2)
                      <> primes (p2:ps)
```

In contrast to typical SQL bindings in other languages, the programmer is not forced to learn the syntax and semantics of another query language (SQL) but formulates the queries in the same programming language (Curry) in a type-safe way. Moreover, typical programming errors that occur in application programs when SQL queries are constructed and passed as strings (a typical error[11] destroying the security of web-based systems [19]) are avoided.

6 Conclusions

We have proposed a new approach to deal with dynamic predicates in functional logic programs. It is based on the idea to separate the update and access to dynamic predicates. Updates can only be performed on the top-level in the I/O monad in order to ensure a well-defined sequence of updates. The access to dynamic predicates is initiated also in the I/O monad in order to get a well-defined set of visible facts for dynamic predicates. However, the actual access can be done at any execution time since the visibility of facts is controlled by time stamps. This is important in the presence of an advanced operational semantics (demand-driven evaluation) where the actual sequence of evaluation steps is difficult to determine in advance.

Furthermore, dynamic predicates can be also persistent so that their definitions are externally stored and recovered when programs are restarted. This persistence model is also supported by a transaction concept in order to pro-

[11] CERT Vulnerabilty Note VU#282403, `http://www.kb.cert.org/vuls/id/282403`

vide the concurrent execution of processes working on the same data. We have sketched an implementation of this concept in a Prolog-based compiler which is freely available with the current release of PAKCS [15].

Although the use of our concept is quite simple (one has to learn only three basic I/O actions), it is quite powerful at the same time since the applications of logic programming to declarative knowledge management can be directly implemented with this concept. We have used this concept in practice to implement a bibliographic database system and obtained quite satisfying results. The loading of the database containing almost 10,000 bibliographic entries needs only a few milliseconds, and querying all facts is also performed in milliseconds due to the fact that they are stored in main memory.

For future work, we want to test this concept in larger applications. Furthermore, we intend to implement this concept by the use of a relational database instead of the current file-based implementation. In this case, the extensions of Section 5 become useful in order to express larger queries that can be directly solved by the database system.

Acknowledgements

The author is grateful to Bernd Braßel and Sebastian Fischer for fruitful discussions related to this work.

References

[1] J.M. Almendros-Jiménez and A. Becerra-Terón. A safe relational calculus for functional logic deductive databases. *Electronic Notes in Theoretical Computer Science*, 86(3), 2003.

[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. 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.

[4] S. Antoy and M. Hanus. Functional logic design patterns. In *Proc. of the 6th International Symposium on Functional and Logic Programming (FLOPS 2002)*, pages 67–87. Springer LNCS 2441, 2002.

[5] B. Braßel, M. Hanus, and F. Huch. Encapsulating non-determinism in functional logic computations. In *Proc. 13th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2004)*, pages 74–90, Aachen (Germany), 2004. Technical Report AIB-2004-05, RWTH Aachen.

[6] B. Bringert and A. Höckersten. HaskellDB improved. In *Proc. of the ACM SIGPLAN 2004 Haskell Workshop*, pages 108–115. ACM Press, 2004.

[7] J. Correas, J.M. Gómez, M. Carro, D. Cabeza, and M. Hermenegildo. A generic persistence model for (C)LP systems (and two useful implementations). In *Proc. of the Sixth International Symposium on Practical Aspects of Declarative Languages (PADL'04)*, pages 104–119. Springer LNCS 3057, 2004.

[8] S.K. Das. *Deductive Databases and Logic Programming*. Addison-Wesley, 1992.

[9] R. Echahed and W. Serwe. Combining mobile processes and declarative programming. In *Proc. of the 1st International Conference on Computation Logic (CL 2000)*, pages 300–314. Springer LNAI 1861, 2000.

[10] H. Gallaire and J. Minker, editors. *Logic and Databases*, New York, 1978. Plenum Press.

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

[12] 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.

[13] 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.

[14] 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.

[15] M. Hanus, S. Antoy, 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/`, 2004.

[16] M. Hanus and F. Huch. An open system to support web-based learning. In *Proc. 12th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2003)*, pages 269–282. Technical Report DSIC-II/13/03, Universidad Politécnica de Valencia, 2003.

[17] 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.

[18] M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.8). Available at `http://www.informatik.uni-kiel.de/~curry`, 2003.

[19] S.H. Huseby. *Innocent Code: A Security Wake-Up Call for Web Programmers*. Wiley, 2003.

[20] D. Leijen and E. Meijer. Domain specific embedded compilers. In *Proceedings of the 2nd Conference on Domain-Specific Languages (DSL'99)*, pages 109–122. ACM SIGPLAN Notices 35(1), 1999.

[21] T.G. Lindholm and R.A. O'Keefe. Efficient implementation of a defensible semantics for dynamic Prolog code. In *Proc. Fourth International Conference on Logic Programming (Melbourne)*, pages 21–39. MIT Press, 1987.

[22] J. Lloyd. Programming in an integrated functional and logic language. *Journal of Functional and Logic Programming*, (3):1–49, 1999.

[23] F. López-Fraguas and J. Sánchez-Hernández. TOY: A multiparadigm declarative system. In *Proc. of RTA'99*, pages 244–247. Springer LNCS 1631, 1999.

[24] L. Naish. Automating control for logic programs. *Journal of Logic Programming (3)*, pages 167–183, 1985.

[25] S.L. Peyton Jones and J. Hughes. Haskell 98: A non-strict, purely functional language. `http://www.haskell.org`, 1999.

[26] 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.

[27] P. Wadler. How to declare an imperative. *ACM Computing Surveys*, 29(3):240–263, 1997.