Functional Logic Programs with Dynamic Predicates*

– Extended Abstract –

Michael Hanus

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

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 [10] 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 [12] or programming dynamic web pages [13].

However, one of the traditional application areas of logic programming is not yet sufficiently covered in existing functional logic languages: the combination 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 [7, 9]) since externally stored relations

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

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 [24], 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 [18] 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 [21]. 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) evaluation strategies where the exact order of evaluation is difficult to determine in advance [2, 11].

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

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 [6], 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 [17] 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 [8] 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 [16] 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 contains a description of our proposal to integrate dynamic predicates into functional logic languages. Section 3 sketches a concrete implementation of this concept and Section 4 contains our conclusions. We assume familiarity with the concepts of functional logic programming [10] and Curry [11, 16].

[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 [19], Mercury [23], or Toy [20].

2 Dynamic Predicates

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

2.1 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. However, in contrast to predicates that are explicitly defined in a program, 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
```

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` (the type of constraints), 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 [24]. Thus, we provide the following predefined operations:

```
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 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 (we use Haskell's "`do`" notation [22] in the following) asserts a new fact `(prime 2)` and retrieves its contents by unifying the logical variable `x` with the value `2`:[4]

```
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 [5, 15]:

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

`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.[5] 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
```

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

[5] `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.

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.

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 3).

2.2 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`.[6] 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 (which is not described in this extended abstract).

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 Implementation

In order to test our concept and to provide a reasonable implementation, we have implemented it in the PAKCS implementation of Curry [14]. The system compiles Curry programs into Prolog by transforming pattern matching into

[6] 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.

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 [18]). 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.

4 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. We have sketched an implementation of this concept in a Prolog-based compiler which is freely available with the current release of PAKCS [14].

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.

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*, Vol. 86, No. 3, 2003.
2. S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. *Journal of the ACM*, Vol. 47, No. 4, pp. 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)*, pp. 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)*, pp. 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)*, pp. 74–90, Aachen (Germany), 2004. Technical Report AIB-2004-05, RWTH Aachen.
6. 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)*, pp. 104–119. Springer LNCS 3057, 2004.
7. S.K. Das. *Deductive Databases and Logic Programming*. Addison-Wesley, 1992.
8. R. Echahed and W. Serwe. Combining Mobile Processes and Declarative Programming. In *Proc. of the 1st International Conference on Computation Logic (CL 2000)*, pp. 300–314. Springer LNAI 1861, 2000.
9. H. Gallaire and J. Minker, editors. *Logic and Databases*, New York, 1978. Plenum Press.

10. M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, Vol. 19&20, pp. 583–628, 1994.
11. M. Hanus. A Unified Computation Model for Functional and Logic Programming. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages (Paris)*, pp. 80–93, 1997.
12. M. Hanus. A Functional Logic Programming Approach to Graphical User Interfaces. In *International Workshop on Practical Aspects of Declarative Languages (PADL'00)*, pp. 47–62. Springer LNCS 1753, 2000.
13. 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)*, pp. 76–92. Springer LNCS 1990, 2001.
14. 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.
15. M. Hanus and F. Steiner. Controlling Search in Declarative Programs. In *Principles of Declarative Programming (Proc. Joint International Symposium PLILP/ALP'98)*, pp. 374–390. Springer LNCS 1490, 1998.
16. M. Hanus (ed.). Curry: An Integrated Functional Logic Language (Vers. 0.8). Available at `http://www.informatik.uni-kiel.de/~curry`, 2003.
17. D. Leijen and E. Meijer. Domain Specific Embedded Compilers. In *Proceedings of the 2nd Conference on Domain-Specific Languages (DSL'99)*, pp. 109–122. ACM SIGPLAN Notices 35(1), 1999.
18. 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)*, pp. 21–39. MIT Press, 1987.
19. J. Lloyd. Programming in an Integrated Functional and Logic Language. *Journal of Functional and Logic Programming*, No. 3, pp. 1–49, 1999.
20. F. López-Fraguas and J. Sánchez-Hernández. TOY: A Multiparadigm Declarative System. In *Proc. of RTA'99*, pp. 244–247. Springer LNCS 1631, 1999.
21. L. Naish. Automating control for logic programs. *Journal of Logic Programming (3)*, pp. 167–183, 1985.
22. S.L. Peyton Jones and J. Hughes. Haskell 98: A Non-strict, Purely Functional Language. `http://www.haskell.org`, 1999.
23. Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, Vol. 29, No. 1-3, pp. 17–64, 1996.
24. P. Wadler. How to Declare an Imperative. *ACM Computing Surveys*, Vol. 29, No. 3, pp. 240–263, 1997.