2015

Default Rules in Functional Logic Programs

Sergio Antoy
Portland State University, antoys@pdx.edu

Michael Hanus
Institut fûr Informatik

Default Rules in Functional Logic Programs*

– Extended Abstract –

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. In functional logic programs, rules are applicable independently of textual order, i.e., any rule can potentially be used to evaluate an expression. This is similar to logic languages and opposite to functional languages, e.g., Haskell enforces a strict sequential interpretation of rules. However, in some situations it is convenient to express alternatives by means of compact default rules. Although default rules are often used in functional programs, the non-deterministic nature of functional logic programs does not allow to directly transfer this concept from functional to functional logic languages in a meaningful way. In this paper we propose a new concept of default rules for Curry that supports a programming style similar to functional programming while preserving the core properties of functional logic programming, i.e., completeness, non-determinism, and logic-oriented uses of functions. We discuss the basic concept and sketch an initial implementation of it which exploits advanced features of functional logic languages.

1 Motivation

Functional logic languages combine the most important features of functional and logic programming in a single language (see [3, 6] for recent surveys). In particular, the functional logic language Curry [7] extends Haskell with common features of logic programming, i.e., non-determinism, free variables, and constraint solving. Moreover, the amalgamated features of Curry supports new programming techniques, like deep pattern matching through the use of functional patterns, i.e., evaluable functions at pattern positions [1]. As a simple example, consider an operation `isSet` intended to check whether a given list represents a set, i.e., does not contain duplicates. In Curry, we might implement it as follows ("`++`" denotes the concatenation of lists):

```
isSet (_++[x]++_++[x]++_) = False
isSet _                   = True
```

The first rule uses a functional pattern: it returns `False` if the argument matches a list where two identical elements occur. If this is not the case, the second rule

* This material is based in part upon work supported by the National Science Foundation under Grant No. 1317249.

returns `True`. However, according to the Curry's semantics, *all* rules are tried to evaluate an expression. Therefore, the second rule is always applicable to calls of `isSet` so that the expression `isSet [1,1]` will be evaluated to `False` *and* `True`.

The unindented application of the second rule can be avoided by the additional requirement that this rule should be applied only if no other rule is applicable. We call such a rule a *default rule* and mark it by adding the suffix `'default` to the function's name. Thus, if we define `isSet` with the rules

```
isSet (_++[x]++_++[x]++_) = False
isSet'default _           = True
```

then `isSet [1,1]` evaluates only to `False` and `isSet [0,1]` only to `True`.

In the following, we sketch an implementation of default rules in Curry where we assume familiarity with the basic concepts of functional logic programming and Curry (see [3, 6, 7]).

2 Default Rules

Default rules are often used in both functional and logic programming. For instance, the following Haskell function reverses a two-element list and leaves all other lists unchanged:

```
rev2 [x,y] = [y,x]
rev2 xs = xs
```

The second rule is applied only if the first rule is not applicable, which yields the intended semantics. We can avoid the consideration of rule orderings by replacing the second rule with rules for the patterns not matching the first rule:

```
rev2 [x,y] = [y,x]
rev2 [] = []
rev2 [x] = [x]
rev2 (x:y:z:xs) = x:y:z:xs
```

This coding is cumbersome in general and impossible in conjunction with functional patterns, as used in the first rule of `isSet` above, since a functional pattern conceptually may denote an infinite set of standard patterns (e.g., `[x,x]`, `[x,_,x]`, `[_,x,_,x]`,...). Thus, there is no finite complement of some functional patterns.

In Prolog, one often uses the cut operator to implement the behavior of default rules. For instance, `rev2` can be defined as a Prolog predicate as follows:

```
rev2([X,Y],[Y,X]) :- !.
rev2(Xs,Xs).
```

Although this behaves as intended for instantiated lists, the completeness of logic programming is destroyed by the cut operator. For instance, the goal `rev2([],[])` is provable, but Prolog does not compute the answer {`Xs=[],Ys=[]`} for the goal `rev2(Xs,Ys)`.

These examples show that a new concept of default rules is required for functional logic programming if we want to keep the strong properties of the base language, in particular, the completeness of logic-oriented evaluations. To avoid developing a new logic foundation of functional logic programming with

default rules, we try to reuse existing features of functional logic languages. We describe our approach explaining the translation of the default rule for `rev2`. The extension to functional patterns and conditional rules can be done in a similar way.

An operation is defined by a set of "standard" rules and one optional default rule that is applied only if no standard rule is applicable because it do not match or its condition is not satisfiable. For this reason, we translate a default rule into a standard rule by adding the condition that no other rule is applicable. For this purpose, we translate the original non-default rules into "test applicability only" rules where the right-hand side is replaced by a constant (here: the unit value "`()`"):

```
rev2'TEST [x,y] = ()
```

Now we add to the default rule the condition that `rev2'TEST` is not applicable. Since we are interested in the failure of attempts to apply `rev2'TEST`, we use a primitive for encapsulating search to check whether `rev2'TEST` has no value. In functional logic programming, set functions [2] or an operator `allValues` [5] have been proposed for this purpose, which behave similarly to Prolog's `findall` but can be used in a declarative manner. Using these primitives, one could translate the default rule into

```
rev2'DEFAULT xs | isEmpty (allValues (rev2'TEST xs)) = xs
```

Hence, this rule can be applied only if all attempts to apply a non-default rule fail. To complete our example, we add this translated default rule as a further alternative to the non-default rule so that we obtain the definition

```
rev2 [x,y] = [y,x]
rev2 xs | isEmpty (allValues (rev2'TEST xs)) = xs
```

Thanks to the logic features of Curry, one can use this definition also to generate appropriate argument values for `rev2`. For instance, if we evaluate the expression `rev2 xs` with the Curry implementation KiCS2 [4], the search space is finite and computes, among others, the binding {`xs=[]`}. This shows that our concept of default rules is more powerful than existing concepts in functional or logic programming. The actual transformation scheme for default rules is more advanced than sketched above in order to accommodate also functional patterns and conditionals rules and to ensure the optimality of functional logic computations even in the presence of default rules.

3 Examples

To show the advantages of default rules for functional logic programming, we sketch a few more examples. In the classical n-queens puzzle, one must place n queens on a chess board so that no queen can attack another queen. This can be solved by computing some permutation of the list `[1..`n`]`, where the i-th element denotes the row of the queen placed in column i, and check whether this permutation is a safe placement. The latter property can easily be expressed with functional patterns and default rules where the non-default rule fails on a non-safe placement:

```
safe (_++[x]++y++[z]++_) | abs (x-z) == length y + 1 = failed
safe'default xs = xs
```

Hence, a solution can be obtained by computing a safe permutation:

```
queens n = safe (permute [1..n])
```

This example shows that default rules are a convenient way to express negation-as-failure from logic programming. This programming pattern can also be applied to solve the map coloring problem. Our map consists of the states of the Pacific Northwest and a list of adjacent states:

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

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

Furthermore, we define the available colors and an operation that associates (non-deterministically) some color to a state (the infix operator "`?`" denotes a non-deterministic choice between its arguments):

```
data Color = Red | Green | Blue

color x = (x, Red ? Green ? Blue)
```

A map coloring can be computed by an operation `solve` that takes the information about potential colorings and adjacent states as arguments, i.e., we compute correct colorings by evaluating the initial expression

```
solve (map color [WA,OR,ID,BC]) adjacent
```

The operation `solve` fails on a coloring where two states have an identical color and are adjacent, otherwise it returns the coloring:

```
solve (_++[(s1,c)]++_++[(s2,c)]++_) (_++[(s1,s2)]++_) = failed
solve'default cs _ = cs
```

References

1. S. Antoy and M. Hanus. Declarative programming with function patterns. In *Proceedings of the International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'05)*, pages 6–22. Springer LNCS 3901, 2005.
2. 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'09)*, pages 73–82. ACM Press, 2009.
3. S. Antoy and M. Hanus. Functional logic programming. *Communications of the ACM*, 53(4):74–85, 2010.
4. B. Braßel, M. Hanus, B. Peemöller, and F. Reck. KiCS2: A new compiler from Curry to Haskell. In *Proc. of the 20th Int. Workshop on Functional and (Constraint) Logic Programming (WFLP 2011)*, pages 1–18. Springer LNCS 6816, 2011.
5. J. Christiansen, M. Hanus, F. Reck, and D. Seidel. A semantics for weakly encapsulated search in functional logic programs. In *Proc. of the 15th International Symposium on Principle and Practice of Declarative Programming (PPDP'13)*, pages 49–60. ACM Press, 2013.
6. M. Hanus. Functional logic programming: From theory to Curry. In *Programming Logics - Essays in Memory of Harald Ganzinger*, pages 123–168. Springer LNCS 7797, 2013.
7. M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.8.3). Available at `http://www.curry-language.org`, 2012.