Default Rules for Curry

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 contrary 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 [7, 15] for recent surveys). In particular, the functional logic language Curry [17] conceptually extends Haskell with common features of logic programming, i.e., non-determinism, free variables, and constraint solving. Moreover, the amalgamated features of Curry support new programming techniques, like *deep* pattern matching through the use of *functional patterns*, i.e., evaluable functions at pattern positions [4].

For example, suppose that we want to compute two elements x and y in a list l with the property that the distance between the two elements is n, i.e., in l there are $n-1$ elements between x and y. We will use this condition in the n-queens program discussed later. Of course, there may be many pairs of elements in a list statisfying the given condition ("`++`" denotes the concatenation of lists):

```
dist n (_++[x]++zs++[y]++_) | n == length zs + 1 = (x,y)
```

Defining functions by case distinction through pattern matching is a very useful feature. Functional patterns make this feature even more convenient. However, in functional logic languages, this feature is slightly more delicate because of the possibility of functional patterns, which typically stand for an infinite number of

standard patterns, and because there is no textual order among the rules defining a function. The variables in a functional pattern are bound like the variables in ordinary patterns.

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:

```
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. The intent of the second rule is to return `True` if no identical elements occur in the argument. However, according to the semantics of Curry, which ensures completeness w.r.t. finding solutions or values, *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 unintended 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 (in order to avoid a syntactic extension of the base language). 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 this paper we propose a concept for default rules for Curry, define its precise semantics, and discuss implementation options. In the next section, we review the main concepts of functional logic programming and Curry. Our intended concept of default rules is informally introduced in Sect. 3. Some examples showing the convenience of default rules for programming are presented in Sect. 4. In order to avoid the introduction of a new semantics specific to default rules, we define the precise meaning of default rules by transforming them into already known concepts in Sect. 5. Options to implement default rules efficiently are sketched and evaluated in Sect. 6 before we relate our proposal to other work and conclude.

2 Functional Logic Programming and Curry

Before presenting the concept and implementation of default rules in more detail, we briefly review those elements of functional logic languages and Curry that are necessary to understand the contents of this paper. More details can be found in recent surveys on functional logic programming [7, 15] and in the language report [17].

Curry is a declarative multi-paradigm language combining in a seamless way features from functional, logic, and concurrent programming (concurrency is irrelevant as our work goes, hence it is ignored in this paper). The syntax of

Curry is close to Haskell [21], i.e., type variables and names of defined operations usually start with lowercase letters and the names of type and data constructors start with an uppercase letter. $\alpha \to \beta$ denotes the type of all functions mapping elements of type α into elements of type β (where β can also be a functional type, i.e., functional types are "curried"), and the application of an operation f to an argument e is denoted by juxtaposition ("$f\ e$"). In addition to Haskell, Curry allows *free* (*logic*) *variables* in conditions and right-hand sides of rules and expressions evaluated by an interpreter. Moreover, the patterns of a defining rule can be non-linear, i.e., they might contain multiple occurrences of some variable, which is an abbreviation for equalities between these occurrences.

Example 1. The following simple program shows the functional and logic features of Curry. It defines an operation "`++`" to concatenate two lists, which is identical to the Haskell encoding. The second operation, `dup`, returns some list element having at least two occurrences:[3]

```
(++) :: [a] → [a] → [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)

dup :: [a] → a
dup xs | xs == _ ++ [x] ++ _ ++ [x] ++ _
       = x    where x free
```

Function calls can contain free variables. They are evaluated lazily where free variables as demanded arguments are non-deterministically instantiated. Hence, the condition of the rule defining `dup` is solved by instantiating `x` and the anonymous free variables "`_`". This evaluation method corresponds to narrowing [24, 22], but Curry narrows with possibly non-most-general unifiers to ensure the optimality of computations [3].

Note that `dup` is a *non-deterministic operation* since it might deliver more than one result for a given argument, e.g., the evaluation of `dup [1,2,2,1]` yields the values `1` and `2`. Non-deterministic operations, which are interpreted as mappings from values into sets of values [13], are an important feature of contemporary functional logic languages. Hence, there is also a predefined *choice* operation:

```
x ? _  =  x
_ ? y  =  y
```

Thus, the expression "`0 ? 1`" evaluates to `0` and `1` with the value non-deterministically chosen.

Some operations can be defined more easily and directly using *functional patterns* [4]. A functional pattern is a pattern occurring in an argument of the left-hand side of a rule containing defined operations (and not only data constructors and variables). Such a pattern abbreviates the set of all standard patterns

[3] Note that Curry requires the explicit declaration of free variables, as `x` in the rule of `dup`, to ensure checkable redundancy.

to which the functional pattern can be evaluated (by narrowing). For instance, we can rewrite the definition of `dup` as

```
dup (_++[x]++_++[x]++_) = x
```

Functional patterns are a powerful feature to express arbitrary selections in tree structures, e.g., in XML documents [14]. Details about their semantics and a constructive implementation of functional patterns by a demand-driven unification procedure can be found in [4].

Set functions [6] allow the encapsulation of non-deterministic computations in a strategy-independent manner. For each defined function f, $f_{\mathcal{S}}$ denotes the corresponding set function. $f_{\mathcal{S}}$ encapsulates the non-determinism caused by evaluating f except for the non-determinism caused by evaluating the arguments to which f is applied. For instance, consider the operation `decOrInc` defined by

```
decOrInc x = (x-1) ? (x+1)
```

Then "$\texttt{decOrInc}_{\mathcal{S}}$ `3`" evaluates to (an abstract representation of) the set $\{2, 4\}$, i.e., the non-determinism caused by `decOrInc` is encapsulated into a set. However, "$\texttt{decOrInc}_{\mathcal{S}}$ `(2?5)`" evaluates to two different sets $\{1, 3\}$ and $\{4, 6\}$ due to its non-deterministic argument, i.e., the non-determinism caused by the argument is not encapsulated. This property is desirable and essential to define and implement default rules by a transformational approach, as shown in Sect. 5. In the following section, we discuss default rules and their intended semantics.

3 Default Rules: Concept and Informal Semantics

Default rules are often used in both functional and logic programming. In languages in which rules are applied in textual order, such as Haskell and Prolog, losely speaking every rule is a default rule of all the preceding rules. For instance, the following standard Haskell function takes two lists and return the list of corresponding pairs, where excess elements of a longer list are discarded:

```
zip (x:xs) (y:ys) = (x,y) : zip xs ys
zip _      _      = []
```

The second rule is applied only if the first rule is not applicable, i.e., if one of the argument lists is empty. We can avoid the consideration of rule orderings by replacing the second rule with rules for the patterns not matching the first rule:

```
zip (x:xs) (y:ys) = (x,y) : zip xs ys
zip (_:_)  []     = []
zip []     _      = []
```

In general, this coding is cumbersome since the number of additional rules increases if the patterns of the first rule are more complex (e.g., we need three additional rules for the function `zip3` combining three lists). Moreover, this coding might be impossible in conjunction with some functional patterns, as in the first rule of `isSet` above. Some functional patterns conceptually denote an infinite set of standard patterns (e.g., `[x,x]`, `[x,_,x]`, `[_,x,_,x]`,...) and the complement of this set is infinite too.

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

```
zip([X|Xs],[Y|Ys],[(X,Y)|Zs]) :- !, zip(Xs,Ys,Zs).
zip(_,_,[]).
```

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

These examples show that neither the functional style nor the logic style of default rules is suitable for functional logic programming. The functional style, based on textual order, curtails non-determinism. The logic style, based on the *cut* operator, destroys the completeness of some computations. Thus, 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, a simple to use non-determinism and the completeness of logic-oriented evaluations. Before presenting the exact definition of default rules, we introduce them informally and discuss their intended semantics.

We intend to extend a "standard" function definition by one default rule. Hence, a function definition with a default rule has the following form ($\overline{o_k}$ denotes a sequence of objects $o_1 \ldots o_k$):[4]

$f\ \overline{t_k^1}$ `|` c_1 `=` e_1
$\vdots$
$f\ \overline{t_k^n}$ `|` c_n `=` e_n
f`'default`$\ \overline{t_k^{n+1}}$ `|` c_{n+1} `=` e_{n+1}

We call the first n rules *standard rules* and the final rule the *default rule* of f. Informally, the default rule is applied only if no standard rule is applicable, where a rule is applicable if the pattern matches and the condition is satisfied. Hence, an expression $e = f\ \overline{s_k}$, where $\overline{s_k}$ are expressions, is evaluated as follows:

1. If there is a standard rule whose left-hand side matches e and the condition is satisfied (i.e., evaluable to `True`), the default rule is ignored to evaluate e.
2. If no standard rule can be applied, the default rule is used to evaluate e.
3. If some argument is non-deterministic, the previous points apply independently for each non-deterministic choice of the combination of arguments. In particular, if an argument is a free variable, it is non-deterministically instantiated to all its possible values before deciding whether the default rule is chosen.

As usual in a non-strict language like Curry, arguments of an operation application are evaluated as they are demanded by the operation's pattern matching and condition. However, any non-determinism or failure during argument evaluation is not passed inside the condition evaluation. A precise definition of "inside"

[4] We consider only conditional rules since an unconditional rule can be regarded as a conditional rule with condition `True`.

is in [6, Def. 3]. This behavior is quite similar to set functions to encapsulate internal non-determinism. Therefore, we will exploit set functions to implement default rules.

Before discussing the advantages and implementation of default rules, we explain and motivate the intended semantics of our proposal. First, it should be noted that this concept distinguishes non-determinism outside and inside a rule application. This difference is irrelevant in purely functional programming but essential in functional logic programming.

Example 2. Consider the operation `zip` defined with a default rule:

```
zip (x:xs) (y:ys) = (x,y) : zip xs ys
zip'default _ _   = []
```

Since the standard rule is applicable to `zip [1] [2]`, the default rule is ignored so that this expression is solely reduced to `(1,2):zip [] []`. Since the standard rule is not applicable to `zip [] []`, the default rule is applied and yields the value `[]`. Altogether, the only value of `zip [1] [2]` is `[(1,2)]`. However, if some argument has more than one value, we use the evaluation principle above for each combination. Thus, the call `zip ([1] ? []) [2]` yields the two values `[(1,2)]` and `[]`.

These considerations are even more relevant if the evaluation of the condition might be non-deterministic, as the following example shows.

Example 3. Consider an operation to look up values for keys in an association list:

```
lookup key assoc   | assoc == (_ ++ [(key,val)] ++ _)
                   = Just val          where val free
lookup'default _ _ = Nothing
```

Note that the condition of the standard rule can be evaluated in various ways. In particular, it can be evaluated (non-deterministically) to `True` and `False` for a fixed association list and key. Therefore, using if-then-else (or an `otherwise` branch as in Haskell) instead of the default rule might lead to unintended results.

If we evaluate `lookup 2 [(2,14),(3,17),(2,18)]`, the condition of the standard rule is satisfiable so that the default rule is ignored. Since the condition has the two solutions $\{\texttt{val} \mapsto 14\}$ and $\{\texttt{val} \mapsto 18\}$, we yield the values `Just 14` and `Just 18`. If we evaluate `lookup 2 [(3,17)]`, the condition of the standard rule is not satisfiable but the default rule is applicable so that we obtain the result `Nothing`.

On the other hand, non-deterministic arguments might trigger different rules to be applied. Consider the expression `lookup (2 ? 3) [(3,17)]`. Since the non-determinism in the arguments leads to independent evaluations of `lookup 2 [(3,17)]` and `lookup 3 [(3,17)]`, we obtain the results `Nothing` and `Just 17`.

Similarly, free variables as arguments might lead to independent results since free variables are equivalent to non-deterministic values [5]. For instance, the expression `lookup 2 xs` yields the value `Just v` with the binding $\{\texttt{xs} \mapsto \texttt{(2,v):\_}\}$,

but also the value `Nothing` with the binding $\{$`xs` $\mapsto$ `[]`$\}$ (as well as many other solutions).

The latter desirable property has also implications for the handling of failures occurring when arguments are evaluated. For instance, consider the expression `lookup 2 failed` (where `failed` is a predefined operation which always fails whenever it is evaluated). Because the evaluation of the condition of the standard rule demands the evaluation of `failed` and the subsequent failure comes from "outside" the condition, the entire expression evaluation fails instead of returning the value `Nothing`. This is motivated by the fact that we need the value of the association list in order to check the satisfiability of the condition and, thus, to decide the applicability of the standard rule, but this value is not available.

Example 4. To see the consequences of an alternative design decision, consider the following contrived definition of an operation that checks whether its argument is the unit value `()` (which is the only value of the unit type):

```
isUnit x | x == () = True
isUnit'default _   = False
```

In our proposal, the evaluation of "`isUnit failed`" fails. In an alternative design (like Prolog's if-then-else construct), one might skip any failure during condition checking and proceed with the next rule. In this case, we would return the value `False` for the expression `isUnit failed`. This is quite disturbing since the (deterministic!) operation `isUnit`, which has only one possible input value, could return two values: `True` for the call `isUnit ()` and `False` for the call `isUnit failed`. Moreover, if we call this operation with a free variable, like `isUnit x`, we obtain the single binding $\{$`x` $\mapsto$ `()`$\}$ and value `True` (since free variables are never bound to failures). Thus, either our semantics would be incomplete for logic computations or we compute too many values. In order to get a consistent behavior, we require that failures of arguments demanded for condition checking lead to failures of evaluations.

4 Examples

To show the applicability and convenience of default rules for functional logic programming, we sketch a few more examples in this section.

Example 5. Default rules are important in combination with functional patterns, since functional patterns denote an infinite set of standard patterns which often has no finite complement. Consider again the operation `lookup` as introduced in Example 3. With functional patterns and default rules, this operation can be conveniently defined:

```
lookup key (_ ++ [(key,val)] ++ _) = Just val
lookup'default _  _                = Nothing
```

Example 6. Functional patterns are also useful to check the deep structure of arguments. In this case, default rules are useful to express in an easy manner

that the check is not successful. For instance, consider an operation that checks whether a string contains a float number (without an exponent but with an optional minus sign). With functional patterns and default rules, the definition of this predicate is easy:

```
isFloat (("-" ? "") ++ n1 ++ "." ++ n2)
        | (all isDigit n1 && all isDigit n2) = True
isFloat'default _ = False
```

Example 7. 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]++zs++[y]++_) | abs (x-y) == length zs + 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.

Example 8. 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:

```
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
```

5 Transformational Semantics

In order to define a precise semantics of default rules, one could extend an existing logic foundation of functional logic programming (e.g., [13]) to include a meaning of default rules. This approach has been partially done in [19] but without considering the different sources of non-determinism (inside, outside) which is important for our intended semantics, as discussed in Sect. 3. Fortunately, the semantic aspects of these issues have already been discussed in the context of encapsulated search [6, 11] so that we can put our proposal on these foundations. Hence, we do not develop a new logic foundation of functional logic programming with default rules, but we provide a transformational semantics, i.e., we specify the meaning of default rules by a transformation into existing constructs of functional logic programming.

We start the description of our transformational approach by explaining the translation of the default rule for `zip`. A default rule is applied only if no standard rule is applicable (because the rule's pattern does not match the argument or the rule's condition is not satisfiable). Hence, we translate a default rule into a regular rule by adding the condition that no other rule is applicable. For this purpose, we generate from the original standard rules a set of "test applicability only" rules where the right-hand side is replaced by a constant (here: the unit value "`()`"). Thus, the single standard rule of `zip` produces the following new rule:

```
zip'TEST (x:xs) (y:ys) = ()
```

Now we have to add to the default rule the condition that `zip'TEST` is not applicable. Since we are interested in the failure of attempts to apply `zip'TEST` to the actual argument, we have to check that this application has no value. Furthermore, non-determinism and failures in the evaluation of actual arguments must be distinguished from similar outcomes caused by the evaluation of the condition.

All these requirements call for the encapsulation of a search for values of `zip'TEST` where "inside" and "outside" non-determinism are distinguished and handled differently. Fortunately, set functions [6] (as sketched in Sect. 2) provide an appropriate solution to this problem. Since set functions have a strategy-independent denotational semantics [11], we will use them to specify and implement default rules. Using set functions, one could translate the default rule into

```
zip xs ys | isEmpty (zip'TEST_S xs ys) = []
```

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

```
zip'TEST (x:xs) (y:ys) = ()

zip (x:xs) (y:ys) = (x,y) : zip xs ys
zip xs ys | isEmpty (zip'TEST_S xs ys) = []
```

Thanks to the logic features of Curry, one can use this definition also to generate appropriate argument values for `zip`. For instance, if we evaluate the equation `zip xs ys == []` with the Curry implementation KiCS2 [10], the search space is finite and computes, among others, the solution {`xs=[]`}.

Unfortunately, this scheme does not yield the best code to ensure optimal computations. To understand the potential problem, consider the following function:

```
f 0 1 = 1
f _ 2 = 2
```

Intuitively, the best strategy to evaluate a call to `f` is a case distinction on the second argument, since its value is demanded by both rules. Formally, `f` is an *inductively sequential* function [1] since the rules can be organized in a structure called "definitional tree" [1] that expresses the following pattern matching strategy:

1. Evaluate the second argument (to head normal form).
2. If its value is `2`, apply the second rule.
3. If its value is `1`, evaluate the first argument and try to apply the first rule.
4. Otherwise, no rule is applicable.

In particular, if `loop` denotes a non-terminating operation, the call `f loop 2` evaluates to `2`. This is in contrast to Haskell [21] which performs pattern matching from left to right so that Haskell loops on this call. This strategy has been extended to functional logic programming (*needed narrowing* [3]) and is known to be optimal for inductively sequential programs. It is also used in Curry but extended to overlapping rules in order to cover general functional logic programs.

Now consider the following default rule for `f`:

```
f'default _ x = x
```

If we apply our transformation scheme sketched above, we obtain the following Curry program:

```
f'TEST 0 1 = ()
f'TEST _ 2 = ()

f 0 1 = 1
f _ 2 = 2
f x y | isEmpty (f'TEST_S x y) = y
```

As a result, the definition of `f` is no longer inductively sequential since the left-hand sides of the first and third rule overlap. Since there is no argument demanded by all rules of `f`, the rules could be applied independently. In fact, the Curry implementation KiCS2 [10] loops on the call `f loop 2` (since it tries to evaluate the first argument in order to apply the first rule), whereas it yields the result `2` without the default rule.

To avoid this undesirable behavior when adding default rules, we could try to use the same strategy for the standard rules and the test in the default rule. This can be done by translating the original standard rules into an auxiliary operation and redefining the original operation into one that either applies the

standard rules or the default rules. For our example, we transform the definition of `f` (with the default rule) into the following functions:

```
f'TEST 0 1 = ()              f'INIT 0 1 = 1
f'TEST _ 2 = ()              f'INIT _ 2 = 2
f'DFLT x y | isEmpty (f'TEST_S x y) = y
f x y = f'INIT x y ? f'DFLT x y
```

Now, both `f'TEST` and `f'INIT` are inductively sequential so that the optimal needed narrowing strategy can be applied, and `f` simply denotes a choice (without an argument evaluation) between two expressions that are evaluated optimally. Observe that at most one of these expressions is reducible. As a result, the Curry implementation KiCS2 evaluates `f loop 2` to `2` and does not run into a loop.

The overall transformation of default rules can be described by the following scheme (its simplicity is advantageous to obtain a comprehensible definition of the semantics of default rules). The function definition

$$
\begin{array}{l}
f\ \overline{t_k^1}\ \texttt{|}\ c_1\ \texttt{=}\ e_1 \\
\vdots \\
f\ \overline{t_k^n}\ \texttt{|}\ c_n\ \texttt{=}\ e_n \\
f\texttt{'default}\ \overline{t_k^{n+1}}\ \texttt{|}\ c_{n+1}\ \texttt{=}\ e_{n+1}
\end{array}
$$

is transformed into (where f`'TEST`, f`'INIT`, f`'DFLT` are new function names):

$$
\begin{array}{ll}
f\texttt{'TEST}\ \overline{t_k^1}\ \texttt{|}\ c_1\ \texttt{= ()} & f\texttt{'INIT}\ \overline{t_k^1}\ \texttt{|}\ c_1\ \texttt{=}\ e_1 \\
\vdots & \vdots \\
f\texttt{'TEST}\ \overline{t_k^n}\ \texttt{|}\ c_n\ \texttt{= ()} & f\texttt{'INIT}\ \overline{t_k^n}\ \texttt{|}\ c_n\ \texttt{=}\ e_n \\
\multicolumn{2}{l}{f\texttt{'DFLT}\ \overline{t_k^{n+1}}\ \texttt{|}\ \texttt{isEmpty (}f\texttt{'TEST}_{\mathcal{S}}\ \overline{t_k^{n+1}}\texttt{)}\ \texttt{\&\&}\ c_{n+1}\ \texttt{=}\ e_{n+1}} \\
\multicolumn{2}{l}{f\ \overline{x_k}\ \texttt{=}\ f\texttt{'INIT}\ \overline{x_k}\ \texttt{?}\ f\texttt{'DFLT}\ \overline{x_k}}
\end{array}
$$

Note that the patterns and conditions of the original rules are not changed. Hence, this transformation is also compatible with other advanced features of Curry, like functional patterns, "as" patterns, non-linear patterns, local declarations, etc. Furthermore, if an efficient strategy exists for the original standard rules, the same strategy can be applied in the presence of default rules. This property can be formally stated as follows:

Proposition 1. *Let $\mathcal{R}$ be a program without default rules, and $\mathcal{R}'$ be the same program except that default rules are added to some operations of $\mathcal{R}$. If $\mathcal{R}$ is overlapping inductively sequential, so is $\mathcal{R}'$.*

Proof. Let f be an operation of $\mathcal{R}$. The only interesting case is when a default rule of f is in $\mathcal{R}'$. Operation f of $\mathcal{R}$ produces four different operations of $\mathcal{R}'$: f, f`'DFLT`, f`'INIT`, and f`'TEST`. The first two are overlapping inductively sequential since they are defined by a single rule. The last two are overlapping inductively sequential when f of $\mathcal{R}$ is overlapping inductively sequential since they have the same definitional tree as f modulo a renaming of symbols. □

The above proposition could be tightened a little when operation f is non-overlapping. In this case three of the four operations produced by the transformation are non-overlapping as well. Prop. 1 is important for the efficiency of computations. In overlapping inductively sequential systems, needed redexes exist and can be easily and efficiently computed [2]. If the original system has a strategy that reduces only needed redexes, the transformed system has a strategy that reduces only needed redexes. This ensures that optimal computations are preserved by the transformation regardless of non-determinism.

This result is in contrast to Haskell (or Prolog), where the concept of default rules is based on a sequential testing of rules, which might inhibit optimal evaluation and prevent or limit non-determinism. Hence, our concept of default rules is more powerful than existing concepts in functional or logic programming (see also Sect. 7).

We now relate values computed in the original system to those computed in the transformed system and vice versa. As expected, extending an operation with a default rule preserves the values computed without the default rule.

Proposition 2. *Let $\mathcal{R}$ be a program without default rules, and $\mathcal{R}'$ be the same program except that default rules are added to some operations of $\mathcal{R}$. If e is an expression of $\mathcal{R}$ that evaluates to the value t w.r.t. $\mathcal{R}$, then e evaluates to t w.r.t. $\mathcal{R}'$.*

Proof. Let $f\ \overline{t_k} \to u$ w.r.t. $\mathcal{R}$, for some expression u, a step of the evaluation of e. The only interesting case is when a default rule of f is in $\mathcal{R}'$. By the definitions of f and f`'INIT` in $\mathcal{R}'$, $f\ \overline{t_k} \to f\texttt{'INIT}\ \overline{t_k} \to u$ w.r.t. $\mathcal{R}'$. A trivial induction on the length of the evaluation of e completes the proof. □

The converse of Prop. 2 does not hold because $\mathcal{R}'$ typically computes more values than $\mathcal{R}$—that is the reason why there are default rules. The following statement relates values computed in $\mathcal{R}'$ to values computed in $\mathcal{R}$.

Proposition 3. *Let $\mathcal{R}$ be a program without default rules, and $\mathcal{R}'$ be the same program except that default rules are added to some operations of $\mathcal{R}$. If e is an expression of $\mathcal{R}$ that evaluates to the value t w.r.t. $\mathcal{R}'$, then either e evaluates to t w.r.t. $\mathcal{R}$ or some default rule of $\mathcal{R}'$ is applied in $e \stackrel{*}{\to} t$ in $\mathcal{R}'$.*

Proof. Let A denote an evaluation $e \stackrel{*}{\to} t$ in $\mathcal{R}'$ that never applies default rules. For any operation f of $\mathcal{R}$, the steps of A are of two kinds: (1) $f\ \overline{t_k} \to f\texttt{'INIT}\ \overline{t_k}$ (2) $f\texttt{'INIT}\ \overline{t_k} \to t'$, for some expressions $\overline{t_k}$ and t'. If we remove from A the steps of kind (1) and replace f`'INIT` with f, we obtain an evaluation of e to t in $\mathcal{R}$. □

In Curry, by design, the textual order of the rules is irrelevant. A default rule is a constructive alternative to a certain kind of failure. For these reasons, a single default rule, as opposed to multiple default rules without any order, is conceptually simpler and adequate in practial situations. Nevertheless, a default rule of a function f may invoke an auxiliary function with multiple ordinary rules thus producing the same behavior of multiple default rules of f.

6 Implementation

The implementation of default rules for Curry based on the transformational approach is available as a preprocessor. The preprocessor is integrated into the compilation chain of the Curry systems PAKCS [16] and KiCS2 [10]. In some future version of Curry, one could also add a specific syntax for default rules and transform them in the front end of the Curry system.

The transformation scheme shown in the previous section is mainly intended to specify the precise meaning of default rules (similarly to the specification of the meaning of guards in Haskell [21]). Although this transformation scheme leads to a reasonably efficient implementation, the actual implementation can be improved in various ways. For instance, the generated functions `f'TEST` and `f'INIT` might duplicate some work to check the patterns and the conditions of the standard rules. This can be avoided by a more sophisticated (but less comprehensible) transformation scheme where the common parts of the definitions of `f'TEST` and `f'INIT` are somehow joined into a single function definition.

Further improvements are possible for specific classes of programs. For instance, consider a function where functional patterns are not used and all standard rules are unconditional. If the left-hand side of the default rule overlaps with the left-hand side of some standard rule, one can compute the complement of the patterns of the standard rules and replace the default rule with patterns from this complement that are compatible with the default pattern. Since the complement might be infinite, one has to find a finite representation of it by considering the size of the standard patterns. For instance, the complement of the pattern `((x:xs),(y:ys))` can be represented by the set {`((x:xs),[])`, `([],_)`} (there are also other possible representations). Hence, we can replace the default rule of `zip` (Example 2) by two rules and obtain the definition shown at the beginning of Sect. 3 and avoid the use of any encapsulated search operation (which is usually less efficient than a case distinction).

As a further example, consider the following definition of the Boolean conjunction:

```
and True  True  = True
and'default _ _ = False
```

By computing the complement pattern of the first rule, we obtain the following transformed definition:

```
and True  True  = True
and True  False = False
and False _     = False
```

Note that this definition is inductively sequential so that it can be evaluated with the optimal needed narrowing strategy [3].

Although the computation of pattern complements is expensive in general [18], it can be done with limited efforts in most practical cases, as shown above. For instance, pattern complements can be computed in a constructive manner with definitional trees. One has to construct a definitional tree for the pattern of the default rule, extend it up to the size of all patterns of standard rules,

and remove the overlaps with patterns of standard rules in order to obtain the remaining patterns for the default rule. The precise description of this method requires a number of technical definitions which are omitted from this paper due to space restrictions.

To show the practical advantage of the transformation with pattern complements, we evaluated a few simple operations defined in a typical functional programming style with default rules. For instance, the computation of the last element of a list can be defined with a default rule as follows:

```
last [x] = x
last'default (_:xs) = last xs
```

Our final example extracts all values in a list of optional ("`Maybe`") values:

```
catMaybes []              = []
catMaybes (Just x : xs)   = x : catMaybes xs
catMaybes'default (_:xs) = catMaybes xs
```

Figure 1 shows the run times (in seconds) to evaluate the operations discussed in this section with the different transformation schemes (i.e., the scheme of Sect. 5 and the improvements with pattern complements presented in this section) and different Curry implementations (where "call size" denotes the number of calls to `and` and the lengths of the input lists for the other examples). All benchmarks were executed on a Linux machine (Debian Jessie) with an Intel Core i7-4790 (3.60Ghz) processor and 8GB of memory. The results clearly indicate the advantage of computing pattern complements, in particular for PAKCS, which has a less sophisticated implementation of set functions than KiCS2.

System:	PAKCS [16]				KiCS2 [10]			
Operation:	`zip`	`and`	`last`	`catMaybes`	`zip`	`and`	`last`	`catMaybes`
Call size:	1000	100000	2000	2000	1000000	1000000	100000	1000000
Sect. 5:	3.66	8.46	2.53	2.45	2.72	1.35	0.38	0.40
Sect. 6:	0.01	0.25	0.01	0.01	0.04	0.08	0.01	0.01

Fig. 1. Performance comparison of different schemes for different compilers for some operations discussed in this section.

7 Related Work

In this section, we compare our proposal of default rules for Curry with existing proposals for other rule-based languages.

The functional programming language Haskell [21] has no explicit concept of default rules. Since Haskell applies the rules defining a function sequentially from top to bottom, it is a common practice in Haskell to write a "catch all" rule as a final rule to avoid writing several nearly identical rules (see example `zip` at the beginning of Sect. 3). Thus, our proposal for default rules increases

the similarities between Curry and Haskell. However, our approach is more general, since it also supports logic-oriented computations, and it is more powerful, since it ensures optimal evaluation for inductively sequential standard rules, in contrast to Haskell (as shown in Sect. 5).

Since Haskell applies rules in a sequential manner, it is also possible to define more than one default rule for a function, e.g., where each rule has a different specificity. This cannot be directly expressed with our default rules where at most one default rule is allowed. However, one can obtain the same behavior by introducing a sequence of auxiliary functions where each function has one default rule.

The logic programming language Prolog [12] is based on backtracking where the rules defining a predicate are sequentially applied. Similarly to Haskell, one can also define "catch all" rules as the final rules of predicate definitions. In order to avoid the unintended application of these rules, one has to put "cut" operators in the preceding standard rules. As already discussed in Sect. 3, these cuts are only meaningful for instantiated arguments so that the completeness of logic programming is destroyed. Hence, this kind of default rules can be used only if the predicate is called in a particular mode, in contrast to our approach.

Various encapsulation operators have been proposed for functional logic programs [9] to encapsulate non-deterministic computations in some data structure. Set functions [6] have been proposed as a strategy-independent notion of encapsulating non-determinism to deal with the interactions of laziness and encapsulation (see [9] for details). One can also use set functions to distinguish successful and non-successful computations, similarly to negation-as-failure in logic programming, exploiting the possibility to check result sets for emptiness. When encapsulated computations are nested and performed lazily, it turns out that one has to track the encapsulation level in order to obtain intended results, as discussed in [11]. Thus, it is not surprising that set functions and related operators fit quite well to our proposal. Actually, many explicit uses of set functions in functional logic programming to implement negation-as-failure can be implicitly and more tersely encoded with our concept of default rules, as shown in Examples 7 and 8.

Default rules and negation-as-failure have been also explored in [19, 23] for functional logic programs. In these works, an operator, `fails`, is introduced to check whether every reduction of an expression to a head-normal form is not successful. [19] proposes the use of this operator to define default rules for functional logic programming. However, the authors propose a scheme where the default rule is applied if no standard rule was able to compute a head normal form. This is quite unusual and in contrast to functional programming (and our proposal) where default rules are applied if pattern matches or conditions of standard rules fail, but the computations of the rules' right-hand sides are not taken into account to decide whether a default rule should be applied. The same applies to an early proposal for default rules in an eager functional logic language [20]. Since the treatment of different sources of non-determinism and their interaction were not explored at that time, nested computations with failures are not considered

by these works. As a consequence, the operator `fails` might yield unintended results if it is used in nested expressions. For instance, if we use `fails` instead of set functions to implement the operation `isUnit` defined in Example 4, the evaluation of `isUnit failed` yields the value `False` in contrast to our intended semantics.

Finally, we proposed in [8] to change Curry's rule selection strategy to a sequential one. However, it turned out that this change has drawbacks w.r.t. the evaluation strategy, since formerly optimal reductions are no longer possible in particular cases. For instance, consider the function `f` defined in Sect. 5 and the call `f loop 2`. In a sequential rule selection strategy, one starts by testing whether the first rule is applicable. Since both arguments are demanded by this rule, one might evaluate them from left to right (as done in the implementation [8]) so that one does not terminate. This problem is avoided with our proposal which returns `2` even in the presence of a default rule for `f`. Moreover, the examples presented in [8] can be expressed with default rules in a similar way.

8 Conclusions

We proposed a new concept of default rules for Curry. Default rules are available in many rule-based languages, but a sensible inclusion into a functional logic language is demanding. Therefore, we used advanced features for encapsulating search to define and implement default rules. Thanks to this approach, typical logic programming features, like non-determinism and evaluating functions with unknown arguments, are still applicable with our new semantics. This distinguishes our approach from similar concepts in logic programming which simply cut alternatives.

Our approach can lead to more elegant and comprehensible declarative programs, as shown by several examples in this paper. Moreover, many uses of negation-as-failure, which are often implemented in functional logic programs by complex applications of encapsulation operators, can easily be expressed with default rules.

Since encapsulated search is more costly than simple pattern matching, we have also shown some opportunities to implement default rules by computing pattern complements. For future work it might be interesting to find more general techniques to transform default rules into case distinctions and tests.

Acknowledgements. The authors are grateful to Sandra Dylus for her suggestions to improve this paper. This material is based in part upon work supported by the National Science Foundation under Grant No. 1317249.

References

1. S. Antoy. Definitional trees. In *Proc. of the 3rd International Conference on Algebraic and Logic Programming*, pages 143–157. Springer LNCS 632, 1992.

2. S. Antoy. Optimal non-deterministic functional logic computations. In *6th Int'l Conf. on Algebraic and Logic Programming (ALP'97)*, volume 1298, pages 16–30, Southampton, UK, 9 1997. Springer LNCS.
3. S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *Journal of the ACM*, 47(4):776–822, 2000.
4. 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.
5. S. Antoy and M. Hanus. Overlapping rules and logic variables in functional logic programs. In *Proceedings of the 22nd International Conference on Logic Programming (ICLP 2006)*, pages 87–101. Springer LNCS 4079, 2006.
6. 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.
7. S. Antoy and M. Hanus. Functional logic programming. *Communications of the ACM*, 53(4):74–85, 2010.
8. S. Antoy and M. Hanus. Curry without Success. In *Proc. of the 23rd International Workshop on Functional and (Constraint) Logic Programming (WFLP 2014)*, volume 1335 of *CEUR Workshop Proceedings*, pages 140–154. CEUR-WS.org, 2014.
9. B. Braßel, M. Hanus, and F. Huch. Encapsulating non-determinism in functional logic computations. *Journal of Functional and Logic Programming*, 2004(6), 2004.
10. B. Braßel, M. Hanus, B. Peemöller, and F. Reck. KiCS2: A new compiler from Curry to Haskell. In *Proc. of the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011)*, pages 1–18. Springer LNCS 6816, 2011.
11. 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.
12. P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog - the standard: reference manual*. Springer, 1996.
13. J.C. González-Moreno, M.T. Hortalá-González, F.J. López-Fraguas, and M. Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming*, 40:47–87, 1999.
14. M. Hanus. Declarative processing of semistructured web data. In *Technical Communications of the 27th International Conference on Logic Programming*, volume 11, pages 198–208. Leibniz International Proceedings in Informatics (LIPIcs), 2011.
15. 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.
16. 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/`, 2015.
17. M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.8.3). Available at `http://www.curry-language.org`, 2012.
18. A. Krauss. Pattern minimization problems over recursive data types. In *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming (ICFP'08)*, pages 267–274. ACM Press, 2008.

19. F.J. López-Fraguas and J. Sánchez-Hernández. A proof theoretic approach to failure in functional logic programming. *Theory and Practice of Logic Programming*, 4(1):41–74, 2004.
20. J.J. Moreno-Navarro. Default rules: An extension of constructive negation for narrowing-based languages. In *Proc. Eleventh International Conference on Logic Programming*, pages 535–549. MIT Press, 1994.
21. S. Peyton Jones, editor. *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press, 2003.
22. U.S. Reddy. Narrowing as the operational semantics of functional languages. In *Proc. IEEE Internat. Symposium on Logic Programming*, pages 138–151, Boston, 1985.
23. J. Sánchez-Hernández. Constructive failure in functional-logic programming: From theory to implementation. *Journal of Universal Computer Science*, 12(11):1574–1593, 2006.
24. J.R. Slagle. Automated theorem-proving for theories with simplifiers, commutativity, and associativity. *Journal of the ACM*, 21(4):622–642, 1974.