To be presented at the 23rd International Workshop on
Functional and (Constraint) Logic Programming (WFLP 2014)
September 15–17, 2014, Leucorea, Lutherstadt Wittenberg, Germany

Curry without Success

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. Curry is a successful, general-purpose, functional logic programming language that predefines a singleton type *Success* explicitly to support its logic component. We take the likely-controversial position that without *Success* Curry would be as much logic or more. We draw a short history and motivation for the existence of this type and justify why its elimination could be advantageous. Furthermore, we propose a new interpretation of rule application which is convenient for programming and increases the similarity between the functional component of Curry and functional programming as in Haskell. We outline some related theoretical (semantics) and practical (implementation) consequences of our proposal.

1 Motivation

Recently, we coded a small Curry [16] module to encode and pretty-print JSON formatted documents [14]. The JSON format encodes floating point numbers with a syntax that makes the decimal point optional. Our Curry System prints floating numbers with a decimal point. Thus, integers, which were converted to floats for encoding, were printed as floats, e.g., the integer value 2 was printed as "2.0". We found all those point-zeros annoying and distracting and decided to get rid of them. To avoid messing with the internal representation of numbers, and risking losing information, our algorithm would look for ".0" at the end of the string representation of a number in the JSON document and remove it. In the *List* library, we found a function, *isSuffixOf*, that tells us whether to drop the last two characters, but we did not find a function to drop the last 2 characters. How could we do that?

In the library we found the usual *drop* and *take* functions that work at the beginning of a string s. Hence, we could reverse s, drop 2 characters, and reverse again. We were not thrilled. Or we could take from s the first $n-2$ characters, where n is the length of s. We were not thrilled either. In both cases, conceptually the string is traversed 3 times (probably in practice too) and extraneous functions are invoked. Not a big deal, but there must be a better way. Although the computation is totally *functional*, we started to think *logic*.

Curry has this fantastic feature called *functional patterns* [4]. With it, we could code the following:

```
fix_int (x ++ ".0") = x                                    (1)
```

Now we were thrilled! This is compact, simple and obviously correct. Of course, we would need a rule for cases in which the string representation of a number does not end in ".0", i.e.:

```
fix_int (x ++ ".0") = x
fix_int x = x
```
(2)

Without the last rule *fix_int* would fail on a string such as "2.1". With the last rule the program would be incorrect because *both* rules would be applied for a number that ends in ".0". The latter is a consequence of the design decision that established that the order of the rules in a program is irrelevant—a major departure of Curry from popular functional languages. One of the reasons of this design decision is *Success*.

2 History

Putting it crudely, a functional logic language is a functional language extended with logic variables. The only complication of this extension is what to do when some function f is applied to some unbound logic variable u. There are two options, either to residuate on u or to narrow u. Residuation suspends the application of *f*, and computes elsewhere in the program in hopes that this computation will narrow u so that the suspended application of f can continue. Narrowing instantiates u to values that sustain the computation. For example, given the usual concatenation of lists:

```
[] ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```
(3)

Narrowing u++t, where u is unbound and t is any expression, instantiates u to `[]` and u'`:`us and continues these computations either one at the time or concurrently depending on the control strategy.

In early functional logic languages [1, 17], in the tradition of logic programming, only predicates (as opposed to any function) are allowed to instantiate a logic variable. In the early days of Curry, we were not brave enough. Indiscriminate narrowing, such as that required for (1), which is based on (3), was uncharted territory and we decided that all functions would residuate except a small selected group called *constraints*. These functions are characterized by returning a singleton type called *Success*.

Narrowing has the remarkable property of solving equations [23]. Indeed, the rule in (1) works by solving an equation by narrowing. An application *fix_int*(s), where s is a string, attempts to solve $s = x$++`".0"`. A solution, "the" if any exists, gives the desired result x. Returning *Success* rather than *Boolean*, as the constrained equality does, had the desirable consequence that we would not "solve" an equation by deriving it to *False*, but had the drawback of introducing a new variant of equality, implemented in Curry by the operation "=`:`=", and the undesirable consequence that "some expressions were more equal than others" [19].

As a consequence of our hesitation, narrowing was limited to the arguments of constraints—a successful model well-established by Prolog. However, this model is at odds with a language with a functional component with normal (lazy) order of evaluation. Without functional nesting, there is no easy way to tell whether or not some argument of some constraint should be evaluated. Consider a program to solve the 8-queens puzzle:

```
permute x y = ...  succeed if y is a permutation of x
safe y = ...       succeed if y is a safe placement of queens
```
(4)

A solution of the puzzle is obtained by *permute* $[1..8]$ `y && safe y`, where y is likely a free variable. The constraint *permute* fully evaluates y upon returning even if *safe* may only look at the first two elements and determine that y is not safe.

This prompted the invention of "non-deterministic functions", i.e., a function-like mechanism, that may return more than one value for the same combination of arguments, but is used as an ordinary function. With this idea, Example (4) is coded as:

```
permute x = ...  return any permutation of x
safe y = ...     as before
```
(5)

In this case, a solution of the puzzle is obtained by *safe* y *where* $y =$ *permute* $[1..8]$. Since y is nested inside *safe*, *permute* $[1..8]$ can be evaluated only to the extent needed by its context. A plausible encoding of *permute* is:

```
permute [] = []
permute (x : xs) = nd_insert x (permute xs)
nd_insert x ys = x : ys
nd_insert x (y : ys) = y : nd_insert x ys
```
(6)

The evaluation of *permute* $[1..8]$ produces any permutation of $[1..8]$ *only if* both rules defining *nd_insert* are applied when the second argument is a non-empty list. Thus, the well-established convention of functional languages that the first rule that matches an expression is the only one being fired had to be changed in the design of Curry.

3 Proposed Adjustments

Our modest proposal is to strip the *Success* type of any special meaning. Since *Success* is isomorphic to the *Unit* type, which is already defined in the *Prelude*, probably it becomes redundant. Future versions of the language could keep it for backward compatibility, but deprecate it.

The first consequence of this change puts in question the usefulness of "=:=", the constrained equality. Equations can be solved using the Boolean equality "==" bringing Curry more in line with functional languages. To solve an equation by narrowing, we simply evaluate it using the standard rules defining Boolean equality. For example, below we show these rules for a polymorphic type *List*:

```
[] == [] = True
(x:xs) == (y:ys) = x==y && xs==ys
[] == (_:_) = False
(_:_) == [] = False
```
(7)

However, we certainly want to avoid binding variables with instantiations that derive an equation to *False* since these bindings are not solutions. Avoiding these bindings is achieved with the following operation:

```
solve True = True
```
(8)

and wrapping an equation with *solve*, i.e., to solve $x = y$, we code *solve (*x == y*)*. Nostalgic programmers could redefine "=:=" as:

```
x =:= y = solve (x == y)                                    (9)
```

When an equation occurs in the condition of a rule, the intended behavior is implied, i.e., the rule is fired only when the condition is (evaluates to) *True*.

In a short paragraph above, we find the symbols "=", "==" and "=:=". The first one is the (mathematical) equality. The other two are (computational) approximations of it with subtle differences. Our proposal simplifies this situation by having only "==" as the implementation of "=", as in functional languages, without sacrificing any of Curry's logic aspects.

The second consequence of our proposal is to review the rule selection strategy, i.e., the order, or more precisely its lack thereof, in which rules are fired. We have already hinted at this issue discussing example (2). Every rule that matches the arguments and satisfies the condition of a call is non-deterministically fired. A motivation for the independence of rule order was discussed in example (6). The ability of making non-deterministic choices is essential to functional logic programming, and it must be preserved, but it can be achieved in a different way.

The predefined operation "?" non-deterministically returns either of its arguments. This operation allows us to express non-determinism in a way different from rules with overlapping left-hand sides [3]. For instance, the non-determinism of the operation *nd_insert* in (6) can be moved from the left-hand sides of its defining rules to the right-hand sides as in the following definition:

```
nd_insert x ys = (x : ys) ? nd_insert2 x ys
nd_insert2 x (y : ys) = y : nd_insert x ys                  (10)
```

Indeed, some Curry compilers, like KiCS2 [8], implement this transformation.

The definition of Curry at the time of this writing [16] establishes that the order of the rules defining an operation is irrelevant. The same holds true for the conditions of a rule, except in the case in which the condition type is Boolean, and for flexible case expressions. Our next proposal is to change this design decision of Curry. Although this is somehow independent of our first proposal to remove the *Success* type, it is reasonable to consider both proposals at once since both simplify the use of Curry.

We propose to change the current definition of rule application in Curry as follows. To determine which rule(s) to fire for an application $t = f(t_1, \ldots, t_n)$, where f is an operation and $t_1, \ldots, t_n$ are expressions, use the following strategy:

1. Scan the rules of f in textual order. An unconditional rule is considered as a conditional rule with condition *True*.
2. Fire the first rule whose left-hand side matches the application t and whose condition is satisfied. Ignore any remaining rule.
3. If no rule can be applied, the computation fails.
4. If a combination of arguments is non-deterministic, the previous points are executed 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.

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" is in [6, Def. 3]. This is quite similar to the behavior of set functions to encapsulate internal non-determinism [6]. Apropos, we discuss in Section 5 how to exploit set functions to implement this concept.

Before discussing the advantages and implementation of this concept, we explain and motivate the various design decisions taken in our proposal. First, it should be noted that this concept distinguishes non-determinism outside and inside a rule application. If the condition of a rule has several solutions, this rule is applied if it is the first one with a true condition. Second, the computation proceeds non-deterministically with all the solutions of the condition. For instance, 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 _ _ = Nothing
```
(11)

If we evaluate *lookup* $2\ [(2,14),(3,17),(2,18)]$, the condition of the first rule is solvable. Thus, we ignore the remaining rules and apply only the first rule to evaluate this expression. Since the condition has the two solutions $\{val \mapsto 14\}$ and $\{val \mapsto 18\}$, we yield the values *Just* 14 and *Just* 18 for this expression. Note that this is in contrast to Prolog's if-then-else construct which checks the condition only once and proceeds just with the first solution of the condition. If we evaluate *lookup* $2\ [(3,17)]$, the condition of the first rule is not solvable but the second 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 rule applications (see item 4), this expression leads to independent evaluations of *lookup* $2\ [(3,17)]$ and *lookup* $3\ [(3,17)]$. The first one yields *Nothing*, whereas the second one yields *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 $\{xs \mapsto (2,v)\texttt{:\_}\}$, but also the value *Nothing* with the binding $\{xs \mapsto \texttt{[]}\}$ (as well as many other solutions). Again, this behavior is different from Prolog's if-then-else construct which performs bindings for free variables inside the condition independently of its source. In contrast to Prolog, our design supports completeness in logic-oriented computations even in the presence of if-then-else.

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 first rule fails, 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, but this value is not available.

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 _ = False
```
(12)

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 $\{x \mapsto \texttt{()}\}$ 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.

Changing the meaning of rule selection from an order-independent semantics to a sequential interpretation is an important change in the design of Curry. However, this change is relevant only for a relatively small amount of existing programs. First, most of the operations in a functional logic program are inductively sequential [2], i.e., they are defined by rules where the left-hand sides do not overlap. Hence, the order of the rules does not affect the definition of such operations. Second, rules defined with traditional Boolean guards residuate if they are applied to unknown arguments, i.e., it is usually not intended to apply alternative conditions to a given call. This fits to a sequential interpretation of conditions. Moreover, our proposal supports the use of conditional rules in a logic programming manner with unknown arguments, since this "outside" non-determinism does not influence the sequential condition checking.

Nevertheless, there are also cases where a sequential interpretation of rules is not intended, e.g., in a rule-oriented programming style, which is often used in knowledge-based or constraint programming. Although we argued that one can always translate overlapping patterns into rules with non-overlapping patterns by using the choice operator "`?`", the resulting code might be less readable. Finally, we have to admit that in a declarative language ignoring the order of the rules is more elegant though not always as convenient. Hence, a good compromise would be a compiler pragma that allows to choose between a sequential or an unordered interpretation of overlapping rules.

4 Advantages

In this section we justify through exemplary problems the advantages of the proposed changes.

Example 1. With the proposed semantics, (2) is a simple and obviously correct solution of the problem, discussed in the introduction, of "fixing" the representation of integers in a JSON document.

Example 2. As in the previous example, our proposed semantics is compatible with functional patterns. Hence, (11) can be more conveniently coded as:

```
lookup key (_ ++ [(key,val)] ++ _) = Just val
lookup _ _ = Nothing
```
(13)

Example 3. Consider a *read-eval-print* loop of a functional logic language such as Curry. A top-level expression may contain free variables that are declared by a *free clause* such as in the following example:

```
x ++ y == [1,2,3,4] where x, y free
```
(14)

Of course, the *free clause* is absent if there are no free variables in the top-level expression. The free variables, when present, are easily extracted with a "deep" pattern as follows:

```
breakFree (exp++" where "++wf++" free"))
   = (exp,wf)
breakFree exp
   = (exp,"")
```
(15)

For this code to work, the rules of `breakFree` must be tried in order and the second one must be fired only if the first one fails.

Example 4. Suppose that World Cup soccer scores are represented in either of the following forms:

```
GER _:_ USA
GER 1:0 USA
```
(16)

where the first line represents a game not yet played and the second one a game in which the digits are the goals scored by the adjacent team (a single digit suffices in practice). The following operation parses scores:

```
parse (team1++" _:_ "++team2) = (team1,team2,Nothing)
parse (team1++[' ',x,':',y,' ']++team2)
   | isDigit x && isDigit y
   = (team1,team2, Just (toInt x,toInt y))
parse _ = error "Wrong format!"
```
(17)

Example 5. The *Dutch National Flag* problem [13] has been proposed in a simple form to discuss the termination of rewriting [12]. A formulation in Curry of this simple form is equally simple:

```
dnf (x++[White,Red]++y) = dnf (x++[Red,White]++y)
dnf (x++[Blue,Red]++y) = dnf (x++[Red,Blue]++y)
dnf (x++[Blue,White]++y) = dnf (x++[White,Blue]++y)
```
(18)

However, (18) needs a termination condition to avoid failure. With our proposed semantics, this condition is simply:

```
dnf x = x
```
(19)

With the standard semantics, a much more complicated condition is needed.

5 Implementation

A good implementation of the proposed changes in the semantics of rule selection requires new compilation schemes for Curry. However, an implementation can also be

obtained by a transformation over source programs when existing advanced features of Curry are exploited. This approach provides a reference semantics that avoids explicitly specifying all the details of our proposal, in particular, the subtle interplay between condition solving and non-determinism and failures in arguments. Hence, we define in this section a program transformation that implements our proposed changes within existing Curry systems.

Initially, we discuss the implementation of a single rule with a sequence of conditions, i.e., a program rule of the form

$$\begin{array}{l} l \;\mid\; c_1 \;=\; e_1 \\ \quad\vdots \\ \quad\mid\; c_k \;=\; e_k \end{array} \tag{20}$$

According to our proposal, if the left-hand side l matches a call, the conditions $c_1, \ldots, c_k$ are sequentially evaluated. If c_i is the first condition that evaluates to `True`, all other conditions are ignored so that (20) becomes equivalent to

$$l \;\mid\; c_i = e_i$$

Note that the subsequent conditions are ignored even if the condition c_i also evaluates to *False*. Thus, the standard translation of rules with multiple guards, as defined in the current report of Curry [16], i.e., replacing multiple guards by nested if-then-else constructs, would yield a non-intended semantics. Moreover, non-determinism and failures in the evaluation of actual arguments must be distinguished from similar outcomes caused by the evaluation of the condition, as discussed in Section 3.

All these requirements call for the encapsulation of condition checking where "inside" and "outside" non-determinism are distinguished and handled differently. Fortunately, recent developments for encapsulated search in functional logic programming [6, 10] provide an appropriate solution of this problem. For instance, [10] proposes an encapsulation primitive *allValues* so that the expression $(\mathit{allValues}\ e)$ evaluates to the set of values of e where only internal non-determinism inside e is considered. Thus, we can use the following expression to check a condition c with our intended meaning:[3]

$$\texttt{if notEmpty (allValues (solve } c\texttt{)) then } e_1 \texttt{ else } e_2 \tag{21}$$

According to [10], the meaning of this expression is as follows:

1. Test whether there is some evaluation of c to *True*.
2. If the test is positive, evaluate e_1.
3. If there is no evaluation of c to *True*, evaluate e_2.

The semantics of *allValues* ensures that non-determinism and failures caused by expressions not defined inside c, in particular, parameters of the left-hand side l of the operation, are not encapsulated. The Curry implementations PAKCS [15] and KiCS2 [8] provide set functions [6] instead of *allValues* which allows the implementation of this conditional in a similar way.

[3] [10] defines only an operation *isEmpty*. Hence we assume that *notEmpty* is defined by the rule $\mathit{notEmpty}\ x = \mathit{not}\ (\mathit{isEmpty}\ x)$.

Our expected semantics demands that a rule with a solvable condition be applied for *each* true condition, in particular, with a possible different binding computed by evaluating the condition. To implement this behavior, we assume an auxiliary operation *ifTrue* that combines a condition and an expression. This operation is simply defined by

```
ifTrue True x = x                                                    (22)
```

Then we define the meaning of (20) by the following transformation:

```
l = if notEmpty (allValues (solve c_1))
     then (ifTrue c_1 e_1) else
    ⋮                                                                (23)
    if notEmpty (allValues (solve c_k))
     then (ifTrue c_k e_k) else failed
```

There are obvious simplifications of this general scheme. For instance, if $c_k = \mathit{True}$, as frequently is the case, the last line of (23) becomes e_k.

This transformation scheme is mainly intended as the semantics of sequential condition checking rather than as the final implementation (similarly to the specification of the meaning of guards in Haskell [20]). A sophisticated implementation could improve the actual code. For instance, each condition c_i is duplicated in our scheme. Moreover, it seems that conditions are always evaluated twice. However, this is not the case if a lazy implementation of encapsulated search via *allValues* or set functions is used, as in the Curry implementation KiCS2 [10]. If c_i is the first solvable condition, the emptiness test for $(\mathit{allValues}\ c_i)$ can be decided after computing a *first* solution. In this case, this solution is computed again (and now also all other solutions) in the then-part in order to pass its computed bindings to e_i. Of course, a more primitive implementation might avoid this duplicated evaluation.

Next we consider the transformation of a sequence of rules

```
l_1  r_1
⋮                                                                    (24)
l_k  r_k
```

where each left-hand side l_i is a pattern $f\ p_{i1} \ldots p_{in_i}$ for the same function f and each r_i is a sequence of condition/expression pairs of the form "$\mid c = e$" as shown in (20).[4] We assume that the pattern arguments p_{ij} contain only constructors and variables. In particular, functional patterns have been eliminated by moving them into the condition using the function pattern unification operator "`=:<=`" (as shown in [4]). For instance, rule (1) is transformed into

```
fix_int xs | (x ++ ".0") =:<= xs = x                                 (25)
```

Finally, we assume that subsequent rules with the same pattern (up to variable renaming) are joined into a single rule with multiple guards. For instance, the rules (2) can be joined (after eliminating the functional pattern) into the single rule

[4] In order to handle all rules in a unique manner, we consider an unconditional rule "$l_i = e_i$" as an abbreviation for the conditional rule "`l_i | True = e_i`".

```
fix_int xs
  | (x ++ ".0") =:<= xs = x
  | True               = xs
```
(26)

Now we distinguish the following cases:

- The patterns in the left-hand sides $l_1, \ldots, l_k$ are inductively sequential [2], i.e., the patterns can be organized in a tree structure such that there is always a discriminating (inductive) argument: since there are no overlapping left-hand sides in this case, the order of the rules is not important for the computed results. Therefore, no further transformation is necessary in this case. Note that most functions in typical functional logic programs are defined by inductively sequential rules.
- Otherwise, there might be overlapping left-hand sides so that it is necessary to check all rules in a sequential manner. For this purpose, we put the pattern matching into the condition so that the patterns and conditions are checked together. Thus, a rule like

  $$f\ p_1 \ldots p_n \mid c = e$$

  is transformed into

  $$f\ x_1 \ldots x_n \mid (\backslash p_1 \ldots p_n \texttt{ -> } c\ )\ x_1 \ldots x_n \\ = (\backslash p_1 \ldots p_n \texttt{ -> ifTrue } c\ e)\ x_1 \ldots x_n$$

  where $x_1, \ldots, x_n$ are fresh variables (the extension to rules with multiple conditions is straightforward). Using this transformation, we obtain a list of rules with identical left-hand sides which can be joined into a single rule with multiple guards, as described above.

For instance, the definition of `fix_int` (26) is transformed into

```
fix_int xs =
  if notEmpty (allValues (solve (x++".0"=:<=xs)))
  then (ifTrue (x++".0" =:<= xs) x)
  else xs
```
(27)

For an example of transforming rules with overlapping patterns, consider an operation that reverses a two-element list and leaves all other lists unchanged:

```
rev2 [x,y] = [y,x]
rev2 xs    = xs
```
(28)

According to our transformation, this definition is mapped into (after some straightforward simplifications):

```
rev2 xs =
  if notEmpty (allValues (\[x,y] -> True) xs)
  then (\[x,y] -> [y,x]) xs
  else xs
```
(29)

Thanks to the logic features of Curry, one can also use this definition to generate appropriate argument values for *rev2*. For instance, if we evaluate the expression *rev2 xs*

(where *xs* is a free variable), the Curry implementation KiCS2 [8] has a finite search space and computes the following bindings and values:

```
{xs = []} []
{xs = [x1]} [x1]
{xs = [x1,x2]} [x2,x1]
{xs = (x1:x2:x3:x4)} (x1:x2:x3:x4)
```

As mentioned above, the transformation presented in this section is intended to serve as a reference semantics for our proposed changes and to provide a prototypical implementation. There are various possibilities to improve this implementation. For instance, if the right-hand side expressions following each condition are always evaluable to a value, i.e., to a finite expression without defined operations, the duplication of the code of the condition as well as the potential double evaluation of the first solvable condition can be easily avoided. As an example, consider the following operation that checks whether a string contains a non-negative float number (without an exponent):

```
isNNFloat (f1 ++ "." ++ f2)
          | all isDigit f1 && all isDigit f2 = True
isNNFloat _= False
```
(30)

If c denotes the condition

```
(f1 ++ "." ++ f2) =:<= s &&
all isDigit f1 && all isDigit f2
```
(31)

by functional pattern elimination [4], program (30) is equivalent to

```
isNNFloat s | c = True
isNNFloat _ = False
```
(32)

Applying our transformation, we obtain the following code with the duplicated condition c:

```
isNNFloat s =
  if notEmpty (allValues (solve c))
  then (ifTrue c True)
  else False
```
(33)

Since the expressions on the right-hand side are always values (*True* or *False*), we can put these expressions into the sets computed by *allValues*. Then the check for a solvable condition becomes equivalent to check the non-emptiness of these value sets so that we return non-deterministically some value of this set.[5] This idea can be implemented by the following scheme which does not duplicate the condition and evaluates it only once (the actual code can be simplified but we want to show the general scheme):

```
isNNFloat s =
  if notEmpty s1 then chooseValue s1 else False
 where
  s1 = allValues (ifTrue c True)
```
(34)

Note that this optimization is not applicable if it is not ensured that the right-hand side expressions are always evaluable to values. For instance, consider definition (28) of

[5] The predefined operation *chooseValue* non-deterministically returns some value of a set.

rev2 and the expression *head* (*rev2* $[nv, 0]$), where nv is an expression without a value (e.g., failure or non-termination). With our current transformation (29), we compute the value 0 for this expression. However, the computation of the set of all values of (*rev2* $[nv, 0]$) w.r.t. the first rule defining *rev2* does not yield any set since the right-hand side $[0, nv]$ has no value. This explains our transformation scheme (23) which might look complicated at a first glance.

However, there is another transformation to implement overlapping rules like (28) with our intended semantics. If the rules are unconditional, one can "complete" the missing constructor patterns in order to obtain an inductively sequential definition. For the operation *rev2*, we obtain the following definition:

```
rev2 [x,y] = [y,x]
rev2 [] = []
rev2 [x] = [x]
rev2 (x:y:z:xs) = x:y:z:xs
```
(35)

Since a case can be more efficiently executed than an encapsulated computation, this alternative transformation might lead to larger but more efficient target code.

6 Related Work

Declarative programming languages support the construction of readable and reliable programs by partitioning complex procedures into smaller units—mainly using case distinction by pattern matching and conditional rules. Since we propose a new interpretation of case distinctions for functional logic programs, we compare our proposal with existing ones with similar objectives.

The functional programming language Haskell [20] provides, similarly to Curry, also pattern matching and guarded rules for case distinctions. Our proposal for a new sequential interpretation of patterns increases the similarities between Curry and Haskell. Although Curry provides more features due to the built-in support to deal with non-deterministic and failing computations, our proposal is a conservative extension of Haskell's guarded rules, i.e., it has the same behavior as Haskell when non-determinism and failures do not occur. To see this, consider a program rule with multiple conditions:

$$\begin{array}{l} l \;\texttt{|}\; c_1 \;\texttt{=}\; e_1 \\ \quad\vdots \\ \quad\texttt{|}\; c_k \;\texttt{=}\; e_k \end{array} \tag{36}$$

Since non-deterministic computations do not exist in Haskell and failures lead to exceptions in Haskell, we assume that, if this rule is applied in Haskell to an expression e, there is one condition c_i which evaluates to *True* and all previous conditions $c_1, \ldots, c_{i-1}$ evaluate to *False*. If we consider the same rule translated with the transformation scheme (23), obviously each condition *notEmpty* (*allValues* (*solve* c_j)) reduces to *False* for $j = 1, \ldots, i-1$ and to *True* for $j = i$. Thus, the application of this rule reduces e to (*ifTrue* c_i e_i) and, subsequently, to e_i, as in Haskell.

The logic programming language Prolog [11] also supports pattern matching and, for sequential conditions, an if-then-else construct of the form "c `->` e_1 `;` e_2". Although Prolog can deal, similarly to Curry, with non-deterministic and failing compu-

tations, the if-then-else construct usually restricts the completeness of the search space due to cutting the choice points created by c before executing e_1. Hence, only the first solution of c is used to evaluate e_1. Furthermore, inside and outside non-determinism is not distinguished so that variables outside the condition c might be bound during its evaluation. This has the effect that predicates where if-then-else is used are often restricted to a particular mode. For instance, consider the re-definition of *rev2* (28) as a predicate in Prolog using if-then-else:

```
rev2(Xs,Ys) :- Xs=[X,Y] -> Ys=[Y,X] ; Ys=Xs.        (37)
```

If we try to solve the goal *rev2*(Xs, Ys), Prolog yields the single answer $Xs = [A, B]$, $Ys = [B, A]$. Thus, in contrast to our approach, all other answers are lost.

Various encapsulation operators have been proposed for functional logic programs [7] 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 [7] for details). We 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 [10]. Thus, it is not surprising that set functions and related operators fit quite well to our proposal.

Computations with failures for the implementation of an if-then-else construct and default rules in functional logic programs have been also explored in [18, 22]. In these works, an operator, *fails*, is introduced to check whether every reduction of an expression to a head-normal form is not successful. The authors show that this operator can be used to define a single default rule, but not the more general sequential rule checking of our approach. Moreover, 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 *allValues* to implement the operation *isUnit* defined in (12), the evaluation of *isUnit failed* yields the value *False* in contrast to our intended semantics.

7 Conclusions

We proposed two changes to the current design of Curry. The first one concerns the removal of the type *Success* and the related constraint equality "`=:=`". This simplifies the language since it relieves the programmer from choosing the appropriate equality operator. The second one concerns a strict order in which rules and conditions are tried to reduce an expression. This makes the language design more similar to functional languages like Haskell so that functional programmers will be more comfortable with Curry. Nevertheless, the 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 cuts alternatives.

However, our proposal comes also with some drawbacks. We already mentioned that in knowledge-based or constraint programming applications, a sequential ordering

of rules is not intended. Hence, a compiler pragma could allow the programmer to choose between a sequential or an unordered interpretation of overlapping rules.

A further drawback of our approach concerns the run-time efficiency. We argued that solving "==" equations by narrowing with standard equational rules can replace the constraint equality "=:=". Although this is true from a semantic point of view, the constraint equality operator "=:=" is more efficient from an operational point of view. If x and y are free variables, the equational constraint "x=:=y" is deterministically solved by binding x to y (or vice versa), whereas the Boolean equality "x==y" is solved by non-deterministically instantiating x and y to identical values. The efficiency improvement of performing bindings is well known, e.g., it is benchmarked in [9] for the Curry implementation KiCS2. On the other hand, the Boolean equality "x==y" is more powerful since it can also solve negated conditions, i.e., evaluate "x==y" to *False* by binding x and y to different values.

Hence, for future work it is interesting to find a compromise, e.g., performing variable bindings when "x==y" should be reduced to *True* without any surrounding negations. A program analysis could be useful to detect such situations at compile time.

Finally, the concurrency features of Curry must be revised. Currently, concurrency is introduced by the concurrent conjunction operator "`&`" on constraints. If the constraint type *Success* is removed, other forms of concurrent evaluations might be introduced, e.g., in operators with more than one demanded argument ("==", "+",...), explicit concurrent Boolean conjunctions, or only in the I/O monad similarly to Concurrent Haskell [21].

Despite all the drawbacks, our proposal is a reasonable approach to simplify the design of Curry and make it more convenient for the programmer.

8 Acknowledgments

This material is based upon work partially supported by the National Science Foundation under Grant No. CCF-1317249.

References

1. H. Aït-Kaci and A. Podelski. Towards a meaning of LIFE. In *Proc. of the 3rd Int. Symposium on Programming Language Implementation and Logic Programming*, pages 255–274. Springer LNCS 528, 1991.
2. S. Antoy. Definitional trees. In *Proc. of the 3rd International Conference on Algebraic and Logic Programming*, pages 143–157. Springer LNCS 632, 1992.
3. S. Antoy. Optimal non-deterministic functional logic computations. In *Proceedings of the Sixth International Conference on Algebraic and Logic Programming (ALP'97)*, pages 16–30, Southampton, UK, September 1997. Springer LNCS 1298. Extended version at `http://cs.pdx.edu/~antoy/homepage/publications/alp97/full.pdf`.
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. B. Braßel, M. Hanus, and F. Huch. Encapsulating non-determinism in functional logic computations. *Journal of Functional and Logic Programming*, 2004(6), 2004.
8. 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.
9. B. Braßel, M. Hanus, B. Peemöller, and F. Reck. Implementing equational constraints in a functional language. In *Proc. of the 15th International Symposium on Practical Aspects of Declarative Languages (PADL 2013)*, pages 125–140. Springer LNCS 7752, 2013.
10. 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.
11. P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog - the standard: reference manual*. Springer, 1996.
12. N. Dershowitz. Termination of rewriting. *J. Symb. Comput.*, 3(1/2):69–116, 1987.
13. E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
14. The JSON Data Interchange Standard.
15. 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/`, 2013.
16. M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.8.3). Available at `http://www.curry-language.org`, 2012.
17. J. Lloyd. Programming in an integrated functional and logic language. *Journal of Functional and Logic Programming*, 1999(3):1–49, 1999.
18. 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.
19. G. Orwell. *Animal Farm: A Fairy Story*. Secker and Warburg, London, UK, 1945.
20. S. Peyton Jones, editor. *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press, 2003.
21. S.L. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Proc. 23rd ACM Symposium on Principles of Programming Languages (POPL'96)*, pages 295–308. ACM Press, 1996.
22. J. Sánchez-Hernández. Constructive failure in functional-logic programming: From theory to implementation. *Journal of Universal Computer Science*, 12(11):1574–1593, 2006.
23. J.R. Slagle. Automated theorem-proving for theories with simplifiers, commutativity, and associativity. *Journal of the ACM*, 21(4):622–642, 1974.