Synthesizing Set Functions

Sergio Antoy[1] Michael Hanus[2] Finn Teegen[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,fte}@informatik.uni-kiel.de

Abstract. Set functions are a feature of functional logic programming to encapsulate all results of a non-deterministic computation in a single data structure. Given a function f of a functional logic program written in Curry, we describe a technique to synthesize the definition of the set function of f. The definition produced by our technique is based on standard Curry constructs. Our approach is interesting for three reasons. It allows reasoning about set functions, it offers an implementation of set functions which can be added to any Curry system, and it has the potential of changing our thinking about the implementation of non-determinism, a notoriously difficult problem.

1 Introduction

Functional logic languages, such as Curry and TOY, combine the most important features of functional and logic languages. In particular, the combination of lazy evaluation and non-determinism leads to better evaluation strategies compared to logic programming [2]. However, the combination of these features poses new challenges. In particular, the encapsulation of non-strict non-deterministic computations has not a universally accepted solution so that different Curry systems offer different implementations for it. Encapsulating non-deterministic computations is an important feature for application programming when the task is to show whether some problem has a solution or to compare different solutions in order to compute the best one.

A realistic example of application is Dijkstra's algorithm for the shortest path in a graph [12]. At each iteration, the algorithm selects the "current" node, finds the set of its unvisited neighbors, and calculates their tentative distances through the current node. Defining a function that takes a node and non-deterministically produces a neighboring node is particularly simple. Our work enables us to find the set of neighbors by encapsulating this non-determinism. In this way, a relatively complicated problem becomes simple.

However, encapsulating non-determinism is not straightforward. Let $S(e)$ denote the set of all the values of an expression e. The problem with such an encapsulation operator is the fact that e might share subexpressions which are defined outside $S(e)$. For instance, consider the expression

let x = 0?1 in S(x) (1)

The infix operator "`?`" denotes a non-deterministic choice, i.e., the expression "`0?1`" has *two* values: `0` or `1`. Since the non-determinism of `x` is introduced outside S(`x`), the question arises whether this should be encapsulated. *Strong encapsulation*, which is similar to Prolog's `findall`, requires to encapsulate all non-determinism occurring during the evaluation of the encapsulated expression. In this case, expression (1) evaluates to the set $\{$`0,1`$\}$. As discussed in [8], a disadvantage of strong encapsulation is its dependence on the evaluation strategy. For instance, consider the expression

```
let x = 0?1 in (S(x), x)                                        (2)
```

If the tuple is evaluated from left to right, the first component evaluates to $\{$`0,1`$\}$ but the second component non-deterministically evaluates to the values `0` and `1` so that the expression (2) evaluates to the values `({0,1},0)` and `({0,1},1)`. However, in a right-to-left evaluation of the tuple, `x` is evaluated first to one of the values `0` and `1` so that, due to sharing, the expression (2) evaluates to the values `({0},0)` and `({1},1)`.

To avoid this dependency on the evaluation strategy, *weak encapsulation* of $S(e)$ only encapsulates the non-determinism of e but not the non-determinism originating from expressions created outside e. Thus, *weak encapsulation* produces the result values `({0},0)` and `({1},1)` of (2) independent of the evaluation strategy. Weak encapsulation has the disadvantage that its meaning depends on the syntactic structure of expressions. For instance, the expressions "`let x = 0?1 in` S(`x`)" and "S(`let x = 0?1 in x`)" have different values. To avoid misunderstandings and make the syntactic structure of encapsulation explicit, *set functions* have been proposed [4]. For any function f, there is a set function $f_{\mathcal{S}}$ which computes the set of all the values f for given argument values. The set function encapsulates the non-determinism caused by the definition of f but not non-determinism originating from arguments. For instance, consider the operation

```
double x = x + x                                                (3)
```

The result of `double`$_{\mathcal{S}}$ is always a set with a single element since the definition of `double` does not contain any non-determinism. Thus, `double`$_{\mathcal{S}}$ `(0?1)` evaluates to the two sets $\{$`0`$\}$ and $\{$`2`$\}$.

Although set functions fit well into the framework of functional logic programming, their implementation is challenging. For instance, the Curry system PAKCS [16] compiles Curry programs into Prolog programs so that non-determinism is implemented for free. Set functions are implemented in PAKCS by Prolog's `findall`. To obtain the correct separation of non-determinism caused by arguments and functions, as discussed above, arguments are completely evaluated before the `findall` encapsulation is invoked. Although this works in many cases, there are some situations where this implementation does not deliver any result. For instance, if the complete evaluation of arguments fails or does not terminate, no result is computed even if the set function does not demand the complete argument values. Furthermore, if the set is infinite, `findall` does not terminate even if the goal is only testing whether the set is empty. Thus, the PAKCS implementation of set functions is "too strict."

These problems are avoided by the implementation of set functions in the Curry system KiCS2, which compiles Curry programs into Haskell programs and represents non-deterministic values in tree-like structures [9]. A similar but slightly different representation is used to implement set functions. Due to the interaction of nested non-determinism, the detailed implementation is quite complex so that a simpler implementation is desirable.

In this paper, we propose a new implementation of set functions that can be added to any Curry system. It avoids the disadvantages of existing implementations by synthesizing an explicit definition of a set function for a given function. Depending on the source code of a function, simple or more complex definitions of a set function are derived. For instance, nested set functions require a more complex scheme than top-level set functions, and functions with non-linear right-hand sides require an explicit implementation of the call-time choice semantics.

The paper is structured as follows. In the next section, we review some aspects of functional logic programming and Curry. After the definition of set functions in Sect. 3, we introduce in Sect. 4 plural functions as an intermediate step towards the synthesis of set functions. A first and simple approach to synthesize set functions is presented in Sect. 5 before we discuss in Sect. 6 and 7 the extension to non-linear rules and nested set functions, respectively. Sect. 8 discussed related work before we conclude in Sect. 9.

2 Functional Logic Programming and Curry

We assume familiarity with the basic concepts of functional logic programming [5, 15] and Curry [17]. Therefore, we briefly review only those aspects that are relevant for this paper.

Although Curry has a syntax close to Haskell [22], there is an important difference in the interpretation of rules defining an operation. If there are different rules that might be applicable to reduce an expression, all rules are applied in a non-deterministic manner. Hence, operations might yield more than one result on a given input. Non-deterministic operations, which are interpreted as mappings from values into sets of values [14], are an important feature of contemporary functional logic languages. The archetype of non-deterministic operations is the choice operator "`?`" defined by

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

Typically, this operator is used to define other non-deterministic operations like

```
coin = 0 ? 1
```

Thus, the expression `coin` evaluates to one of the values `0` or `1`. Non-deterministic operations are quite expressive since they can be used to completely eliminate logic variables in functional logic programs, as shown in [3, 11]. Therefore, we ignore logic variables in the formal development below. For instance, a Boolean logic variable can be replaced by the non-deterministic *generator operation* for Booleans defined by

```
aBool = False ? True
```
(4)

Passing non-deterministic operations as arguments, as in the expression `double coin`, might cause a semantical ambiguity. If the argument `coin` is evaluated before calling `double`, the expression has two values, `0` and `2`. However, if the argument `coin` is passed unevaluated to the right-hand side of `double` and, thus, duplicated, the expression has three different values: `0`, `1`, or `2`. These two interpretations are called *call-time choice* and *run-time choice* [19]. Contemporary functional logic languages stick to the call-time choice, since this leads to results which are independent of the evaluation strategy and has the rewriting logic CRWL [14] as a logical foundation for declarative programming with non-strict and non-deterministic operations. Furthermore, it can be implemented by sharing which is already available in implementations of non-strict languages. In this paper, we use a simple reduction relation, equivalent to CRWL, that we sketch without giving all details (which can be found in [20]).

A *value* is an expression without occurrences of function symbols. To cover non-strict computations, expressions can also contain the special symbol $\bot$ to represent *undefined or unevaluated values*. A *partial value* is a value that might contain occurrences of $\bot$. A *partial constructor substitution* is a substitution that replaces variables by partial values. A *context* $\mathcal{C}[\cdot]$ is an expression with some "hole." Then expressions are reduced according to the following reduction relation:

$$\begin{array}{rcll} \mathcal{C}[f\ \sigma(t_1)\ldots\sigma(t_n)] & \to & \mathcal{C}[\sigma(r)] & \text{where } f\ t_1\ldots t_n = r \text{ is a program rule} \\ & & & \text{and } \sigma \text{ a partial constructor substitution} \\ \mathcal{C}[e] & \to & \mathcal{C}[\bot] & \text{where } e \neq \bot \end{array}$$

The first rule models the call-time choice: if a rule is applied, the actual arguments of the operation must have been evaluated to partial values. The second rule models non-strictness by allowing the evaluation of any subexpression to an undefined value (which is intended if the value of this subexpression is not demanded). As usual, $\overset{*}{\to}$ denotes the reflexive and transitive closure of this reduction relation. We also write $e = v$ instead of $e \overset{*}{\to} v$ if v is a (partial) value.

For the sake of simplicity, we assume that programs are already translated into a simple standard form: conditional rules are replaced by if-then-else expressions and the left-hand sides of all operations (except for "`?`") are *uniform* [21] i.e., either the operation is defined by a single rule where all arguments are distinct variables, or in the left-hand sides of all rules only the last (or any other fixed) argument is a constructor with variables as arguments where each constructor of a data type occurs in exactly one rule. Section 3.2 of [21] provides an algorithm for transforming the rules defining a function into a set of uniform rules defining the same function. Uniform rules are not overlapping so that non-determinism is represented by `?`-expressions.

3 Set Functions

The purpose of a set function is to encapsulate only the non-determinism caused by the definition of the corresponding function. Similarly to non-determinism, set functions encapsulate failures only if they are caused by the function's definition. If `failed` denotes a failing computation, i.e., an expression without a value, the expression `double`$_\mathcal{S}$ `failed` has no value (and not the empty set as a value). Since the meaning of failures and nested set functions has not been discussed in [4], Christiansen et al. [10] propose a rigorous denotational semantics for set functions. In order to handle failures and choices in nested applications of set functions, computations are attached with "nesting levels" so that failures caused by different encapsulation levels can be distinguished.

In the following, we use a simpler model of set functions. Formally, set functions return sets. However, for simplicity, our implementation returns multisets, instead of sets, represented as some abstract data type. Converting multisets into sets is straightforward for the representation we choose. If b is a type, $\{b\}$ denotes the type of a set of elements of type b. The meaning of a set function can be defined as follows:

Definition 1. *Given a unary (for simplicity) function* $f :: a \to b$*, the* set function *of* f*,* $f_\mathcal{S} :: a \to \{b\}$*, is defined as follows: For every partial value* t *of type* a*, value* u *of type* b*, and set* U *of elements of type* b*,* $(f\ t) = u$ *iff* $(f_\mathcal{S}\ t) = U$ *and* $u \in U$*.*

This definition accommodates the different aspects of set functions discussed in the introduction. If the evaluation of an expression e leads to a failure or choice but its value is not required by the set function, it does not influence the result of the set function since e can be derived to the partial value $\perp$.

Example 1. Consider the following function:

```
ndconst x y = x ? 1                                                (5)
```

The value of `ndconst`$_\mathcal{S}$ `2 failed` is `{2,1}`, and `ndconst`$_\mathcal{S}$ `(2?4) failed` has the values `{2,1}` and `{4,1}`.

Given a function f, we want to develop a method to synthesize the definition $f_\mathcal{S}$. A difficulty is that f might be composed of other functions whose non-determinism should also be encapsulated by $f_\mathcal{S}$. This can be solved by introducing plural functions, which are described next.

4 Plural Functions

If $f :: b \to c$ and $g :: a \to b$ are functions, their composition $(f \circ g)$ is well defined by $(f \circ g)(x) = f(g(x))$ for all x of type a. However, the corresponding set functions, $f_\mathcal{S} :: b \to \{c\}$ and $g_\mathcal{S} :: a \to \{b\}$, are not composable because their types mismatch—an output of $g_\mathcal{S}$ cannot be an input of $f_\mathcal{S}$. To support the composition of functions that return sets, we need functions that take sets as arguments.[3]

Definition 2. *Let $f :: a \to b$ be a function. We call* plural *function of f any function $f_{\mathcal{P}} :: \{a\} \to \{b\}$ with the following property: for all X and Y such that $f_{\mathcal{P}}\ X = Y$, (1) if $y \in Y$ then there exists some $x \in X$ such that $f\ x = y$ and (2) if $x \in X$ and $f\ x = y$, then $y \in Y$.*

The above definition generalizes to functions with more than one argument. The following display shows both the type and an example of application of the plural function, denoted by "$\texttt{++}_{\mathcal{P}}$", of the usual list concatenation "`++`":

```
(++𝒫) :: {[a]} -> {[a]} -> {[a]}
{[1],[2]} ++𝒫 {[],[3]} = {[1],[1,3],[2],[2,3]}
```
(6)

Plural functions are unique, composable, and cover all the results of set functions (see Appendix A for details). Since plural functions are an important step towards the synthesis of set functions, we discuss their synthesis first. To implement plural functions in Curry, we have to decide how to represent sets (our implementation returns multisets) of elements. An obvious representation are lists. Since we will later consider non-linear rules and also nested set functions where non-determinism of different encapsulation levels are combined, we instead use search trees [8] to represent choices between values. The type of a search tree parameterized over the type of elements can be defined as follows:

```
data ST a = Val a | Fail | Choice (ST a) (ST a)
```

Hence, a search tree is either an expression in head-normal form, i.e., rooted by a constructor symbol, a failure, or a choice between search trees. Although this definition does not enforce that the argument of a `Val` constructor is in head-normal form, this invariant will be ensured by our synthesis method for set functions, as presented below. For instance, the plural function of the operation `aBool` (4) can be defined as

```
aBoolP :: ST Bool
aBoolP = Choice (Val False) (Val True)
```

The plural function of the logical negation `not` defined by

```
not False = True
not True  = False
```
(7)

takes a search tree as an argument so that its definition must match all search tree constructors. Since the matching structure is similar for all operations performing pattern matching on an argument, we use the following generic operation to apply an operation defined by pattern matching to a non-deterministic argument:[4]

[3] The notion of "plural function" is also used in [23] to define a "plural" semantics for functional logic programs. Although the type of their plural functions is identical to ours, their semantics is quite different.

[4] Actually, this operation is the monadic "bind" operation with flipped arguments if `ST` is an instance of `MonadPlus`, as proposed in [13]. Here, we prefer to provide a more direct implementation.

```
applyST :: (a  →  ST b)  →  ST a  →  ST b
applyST f (Val x)        = f x
applyST _ Fail           = Fail
applyST f (Choice x1 x2) = Choice (f ‘applyST‘ x1) (f ‘applyST‘ x2)
```

Hence, failures remain as failures, and a choice in the argument leads to a choice in the result of the operation, which is also called a pull-tab step [1]. Now the plural function of `not` can be defined by (shortly we will specify a systematic translation method)

```
notP :: ST Bool  →  ST Bool
notP = applyST $ \x  →  case x of False  →  Val True
                                  True   →  Val False
```

The synthesis of plural functions for uniform programs is straightforward: pattern matching is implemented with `applyST` and function composition in right-hand sides comes for free.[5] For instance, the plural function of

```
twiceNot x = not (not x)
```

is

```
twiceNotP x = notP (notP x)
```

So far we considered only base values in search trees. If one wants to deal with structured data, like lists of integers, a representation like `ST [Int]` is not appropriate since non-determinism can occur in any constructor of the list, as shown by

```
one23 = (1?2) : ([] ? (3:[]))
```

The expression `one23` evaluates to `[1]`, `[2]`, `[1,3]`, and `[2,3]`. If we select only the head of the list, the non-determinism in the tail does not show up, i.e., `head one23` evaluates to two values `1` and `2`. This demands for a representation of head-normal forms with possible search tree arguments. It can be easily derived for any algebraic data type. The head-normal forms of non-deterministic lists are the usual list constructors where the cons arguments are search trees:

```
data STList a = Nil | Cons (ST a) (ST (STList a))
```

The plural representation of `one23` is

```
one23P :: ST (STList Int)
one23P = Val (Cons (Choice (Val 1) (Val 2))
                   (Choice (Val Nil) (Val (Cons (Val 3) (Val Nil)))))
```

The plural function of `head` is synthesized as

```
headP :: ST (STList a)  →  ST a
headP = applyST $ \xs  →  case xs of Nil       →  Fail
                                     Cons x _  →  x
```

so that `headP one23P` evaluates to `Choice (Val 1) (Val 2)`, as intended.

To provide a precise definition of this transformation, we assume that all operations in the program are uniform (see Sect. 2). The plural transformation $[\![\cdot]\!]_{\mathcal{P}}$ of these kinds of function definitions is defined as follows (where $C_{\mathcal{P}}$ denotes the constructor of the non-deterministic type, like `STList`, corresponding to the

[5] This is a consequence of the fact that `ST` is a functor. A more general treatment of these structures can be found in [6].

original constructor C):

$$[\![f\ x_1 \ldots x_n \texttt{ = } e]\!]_{\mathcal{P}} \quad = \quad f_{\mathcal{P}}\ x_1 \ldots x_n \texttt{ = } [\![e]\!]_{\mathcal{P}}$$

$$\left[\!\!\left[\begin{array}{r} f\ x_1 \ldots x_{n-1}\ (C^1\ x_{11} \ldots x_{1i_1}) \texttt{ = } e_1 \\ \vdots \\ f\ x_1 \ldots x_{n-1}\ (C^n\ x_{n1} \ldots x_{ni_n}) \texttt{ = } e_n \end{array}\right]\!\!\right]_{\mathcal{P}} \quad = \quad \begin{array}{l} f_{\mathcal{P}}\ x_1 \ldots x_{n-1} \texttt{ = applyST \$ } \backslash x \rightarrow \\ \quad \texttt{case } x \texttt{ of} \\ \qquad C^1_{\mathcal{P}}\ x_{11} \ldots x_{1i_1} \rightarrow [\![e_1]\!]_{\mathcal{P}} \\ \qquad \vdots \\ \qquad C^n_{\mathcal{P}}\ x_{n1} \ldots x_{ni_n} \rightarrow [\![e_n]\!]_{\mathcal{P}} \end{array}$$

Note that x_i and x_{jk} have different types in the original and transformed program, e.g., an argument of type `Int` is transformed into an argument of type `ST Int`. Furthermore, expressions occurring in the function bodies are transformed according to the following rules:

$$\begin{array}{rcl} [\![x]\!]_{\mathcal{P}} & = & x \\ [\![C\ e_1 \ldots e_n]\!]_{\mathcal{P}} & = & \texttt{Val (}C_{\mathcal{P}}\ [\![e_1]\!]_{\mathcal{P}} \ldots [\![e_n]\!]_{\mathcal{P}}\texttt{)} \\ [\![f\ e_1 \ldots e_n]\!]_{\mathcal{P}} & = & f_{\mathcal{P}}\ [\![e_1]\!]_{\mathcal{P}} \ldots [\![e_n]\!]_{\mathcal{P}} \\ [\![e_1 \texttt{ ? } e_2]\!]_{\mathcal{P}} & = & \texttt{Choice } [\![e_1]\!]_{\mathcal{P}}\ [\![e_2]\!]_{\mathcal{P}} \\ [\![\texttt{failed}]\!]_{\mathcal{P}} & = & \texttt{Fail} \end{array}$$

The presented synthesis of plural functions is simple and yields compositionality and laziness. Thus, they are a good basis to define set functions, as shown next.

We are not overly concerned about the increase in size of a program's object code when plural functions are synthesized and added to the program. The source code of the plural function, $f_{\mathcal{P}}$, of a function f contains the same structural elements, e.g., pattern matching and nested function calls, as f. Hence the object code of $f_{\mathcal{P}}$ is expected to have a size similar to that of f regardless of the compilation technique. Since only a fraction of the functions of a program require the synthesis of the corresponding plural function, the size of the object code should increase by a factor closer to 1 than to 2.

5 Synthesis of Set Functions: The Simple Way

Plural functions take sets as arguments whereas set functions are applied to standard expressions which might not be evaluated. To distinguish these possibly unevaluated arguments from head-normal forms, we add a new constructor to search trees

```
data ST a = Val a | Uneval a | Fail | Choice (ST a) (ST a)
```

and extend the definition of `applyST` with the rule

```
applyST f (Uneval x) = f x
```

Furthermore, plural functions yield non-deterministic structures which might not be completely evaluated. By contrast, set functions yield sets of values, i.e., completely evaluated elements. In order to turn a plural function into a set function, we have to evaluate the search tree structure into the set of their values. For the sake of simplicity, we represent the latter as ordinary lists. Thus, we need an operation like

```
stValues :: ST a  →  [a]
```

to extract all the values from a search tree. For instance, the expression

```
stValues (Choice (Val 1) (Choice Fail (Val 2)))
```

should evaluate to the list `[1,2]`. This demands for the evaluation of all the values in a search tree (which might be head-normal forms with choices at argument positions) into its complete normal form. We define a type class[6] for this purpose:

```
class NF a where
  nf :: a  →  ST a
```

Each instance of this type class must define a method `nf` which evaluates a given head-normal form into a search tree where all `Val` arguments are completely evaluated. Instances for base types are easily defined:

```
instance NF Int where
  nf x = Val x
```

The operation `nf` is easily extended to arbitrary search trees:[7]

```
nfST :: NF a => ST a  →  ST a
nfST (Val x)        = nf x
nfST (Uneval x)     = x ‘seq‘ nf x
nfST Fail           = Fail
nfST (Choice x1 x2) = Choice (nfST x1) (nfST x2)
```

Now we can define an operation that collects all the values in a search tree (without `Uneval` constructors) into a list by a depth-first strategy:

```
searchDFS :: ST a  →  [a]
searchDFS (Val x)        = [x]
searchDFS Fail           = []
searchDFS (Choice x1 x2) = searchDFS x1 ++ searchDFS x2
```

Thus, failures are ignored and choices are concatenated. Combining these two operations yields the desired definition of `stValues`:

```
stValues :: NF a => ST a  →  [a]
stValues = searchDFS . nfST
```

`NF` instances for structured types can be defined by moving choices and failures in arguments to the root:

```
instance NF a => NF (STList a) where
  nf Nil         = Val Nil
  nf (Cons x xs) = case nfST x of
    Choice c1 c2  →  Choice (nf (Cons c1 xs)) (nf (Cons c2 xs))
    Fail          →  Fail
```

[6] Although the current definition of Curry [17] does not include type classes, many implementations of Curry, like PAKCS, KiCS2, or MCC, support them.

[7] The use of `seq` ensures that the `Uneval` argument is evaluated. Thus, non-determinism and failures in arguments of set functions are not encapsulated, as intended.

```
      y             →  case nfST xs of
        Choice c1 c2  →  Choice (nf (Cons y c1)) (nf (Cons y c2))
        Fail          →  Fail
        ys            →  Val (Cons y ys)
```

For instance, the non-deterministic list value `[1?2]` can be described by the `ST` structure

```
nd01 = Val (Cons (Choice (Val 0) (Val 1)) (Val Nil))
```

so that `stValues nd01` moves the inner choice to the top-level and yields the list

```
[Cons (Val 0) (Val Nil), Cons (Val 1) (Val Nil)]
```

which represents the set {`[0]`, `[1]`}.

As an example for our first approach to synthesize set functions, consider the following operation (from Curry's prelude) which non-deterministically returns any element of a list:

```
anyOf :: [a]  →  a
anyOf (x:xs) = x ? anyOf xs
```

Since set functions do not encapsulate non-determinism caused by arguments, the expression $\texttt{anyOf}_{\mathcal{S}}$ `[0?1,2,3]` evaluates to the sets {`0`, `2`, `3`} and {`1`, `2`, `3`}.

In order to synthesize the set function for `anyOf` by exploiting plural functions, we have to convert ordinary types, like `[Int]`, into search tree types, like `STList Int`, and vice versa. For this purpose, we define two conversion operations for each type and collect their general form in the following type class:[8]

```
class ConvertST a b where
  toValST   :: a  →  b
  fromValST :: b  →  a
```

Instances for base and list types are easily defined:

```
instance ConvertST Int Int where
  toValST   = id
  fromValST = id

instance ConvertST a b => ConvertST [a] (STList b) where
  toValST []     = Nil
  toValST (x:xs) = Cons (toST x) (toST xs)

  fromValST Nil                       = []
  fromValST (Cons (Val x) (Val xs)) = fromValST x : fromValST xs
```

where the operation `toST` is like `toValST` but adds an `Uneval` constructor:

```
toST :: ConvertST a b => a  →  ST b
toST = Uneval . toValST
```

The (informal) precondition of `fromValST` is that its argument is already fully evaluated, e.g., by an operation like `stValues`. Therefore, we define the following operation to translate an arbitrary search tree into the list of its Curry values:

```
fromST :: (ConvertST a b, NF b) => ST b  →  Values a
fromST = map fromValST . stValues
```

[8] Multi-parameter type classes are not yet supported in the Curry systems PAKCS and KiCS2. The code presented here is more elegant, but equivalent, to the actual implementation.

As already mentioned, we use lists to represent multisets of values:

```
type Values a = [a]
```

However, one could also use another (abstract) data type to represent multisets or even convert them into sets, if desired.

Now we have all parts to synthesize a set function: convert an ordinary value into its search tree representation, apply the plural function on it, and translate the search tree back into the multiset (list) of the values contained in this tree. We demonstrate this by synthesizing the set function of `anyOf`.

The uniform representation of `anyOf` performs complete pattern matching on all constructors:

```
anyOf []     = failed
anyOf (x:xs) = x ? anyOf xs
```

We easily synthesize its plural function according to the scheme of Sect. 4:

```
anyOfP :: ST (STList Int) → ST Int
anyOfP = applyST $ \xs →
           case xs of Nil       → Fail
                      Cons x xs → Choice x (anyOfP xs)
```

Finally, we obtain its set function by converting the argument into the search tree and the result of the plural function into a multiset of integers:

```
anyOfS :: [Int] → Values Int
anyOfS = fromST . anyOfP . toST
```

The behavior of our synthesized set function is identical to their original definition, e.g., `anyOfS [0?1,2,3]` evaluates to the lists `[0,2,3]` and `[1,2,3]`, i.e., non-determinism caused by arguments is not encapsulated. This is due to the fact that the evaluation of arguments, if they are demanded inside the set function, are initiated by standard pattern matching so that a potential non-deterministic evaluation leads to a non-deterministic evaluation of the synthesized set function.

In contrast to the strict evaluation of set functions in PAKCS, as discussed in the introduction, our synthesized set functions evaluate their arguments lazily. For instance, the set function of `ndconst` defined in Example 1 is synthesized as follows:

```
ndconstP :: ST Int → ST Int → ST Int
ndconstP nx ny = Choice nx (Val 1)

ndconstS :: Int → Int → Values Int
ndconstS x y = fromST (ndconstP (toST x) (toST y))
```

Since the second argument of `ndconstS` is never evaluated, the expression `ndconstS 2 failed` evaluates to `[2,1]` and `ndconstS (2?4) (3?5)` yields the lists `[2,1]` and `[4,1]`. The set function implementation of PAKCS fails on the first expression and yields four results on the second one. Hence, our synthesized set function yields better results than PAKCS, in the sense that it is more complete and avoids duplicated results. Moreover, specific primitive operations, like `findall`, are not required.

The latter property is also interesting from another point of view. Since PAKCS uses Prolog's `findall`, the evaluation strategy is fixed to a depth-first search strategy implemented by backtracking. Our implementation allows more flexible search strategies by modifying the implementation of `stValues`. Actually, one can generalize search trees and `stValues` to a monadic structure, as done in [7, 13], to implement various strategies for non-deterministic programming.

A weak point of our current synthesis is the handling of failures. For instance, the evaluation of `anyOfS [failed,1]` fails (due to the evaluation of the first list element) whereas $\texttt{anyOf}_{\mathcal{S}}$ `[failed,1]` $= \{\texttt{1}\}$ according to Def. 1. To correct this incompleteness, failures resulting from argument evaluations must be combined with result sets. This can be done by extending search trees and distinguishing different sources of failures, but we omit it here since a comprehensive solution to this issue will be presented in Sect. 7 when nested applications of set functions are discussed.

6 Adding Call-Time Choice

We have seen in Sect. 1 that the expression `double (0?1)` should evaluate to the values `0` or `2` due to the call-time choice semantics. Thus, the set function of

```
double01 :: Int
double01 = double (0?1)
```

should yield the multiset $\{\texttt{0},\texttt{2}\}$. However, with the current synthesis, the corresponding set function yields the list `[0,1,1,2]` and, thus, implements the run-time choice. The problem arises from the fact that the non-deterministic choice in the synthesized plural function

```
double01P :: ST Int
double01P = doubleP (Choice (Val 0) (Val 1))
```

is duplicated by `doubleP`. In order to implement the call-time choice, the same decision (left or right choice) for both duplicates has to be made. Instead, the search operation `searchDFS` handles these choices independently and is unaware of the duplication.

To tackle this problem, we follow the idea implemented in KiCS2 [9] and extend our search tree structure by identifiers for choices (represented by the type `ID`) as follows:

```
data ST a = Val a | Uneval a | Fail | Choice ID (ST a) (ST a)
```

The changes to previously introduced operations on search trees, like `applyST` or `nfST`, are minimal and straightforward as we only have to keep a choice's identifier in their definitions. The most significant change occurs in the search operation. As shown in [9], the call-time choice can be implemented by storing the decision for a choice, when it is made for the first time, during the traversal of the search tree and looking it up later when encountering the same choice again. We introduce the type

```
data Decision = Left | Right
```

for decisions and use an association list[9] as an additional argument to the search operation to store such decisions. The adjusted depth-first search then looks as follows:

[9] Of course, one can replace such lists by more efficient access structures.

```
searchDFS :: [(ID,Decision)] → ST a → [a]
searchDFS _ (Val x)          = [x]
searchDFS _ Fail             = []
searchDFS m (Choice i x1 x2) = case lookup i m of
  Nothing     → searchDFS ((i,Left):m) x1 ++
                 searchDFS ((i,Right):m) x2
  Just Left   → searchDFS m x1
  Just Right  → searchDFS m x2
```

When extracting all the values from a search tree, we initially pass an empty list to the search operation since no decisions have been made at that point:

```
stValues :: NF a => ST a → [a]
stValues = searchDFS [] . nfST
```

Finally, we have to ensure that the choices occurring in synthesized plural functions are provided with unique identifiers. To this end, we assume a type `IDSupply` that represents an infinite set of such identifiers along with the following operations:

```
initSupply               :: IDSupply
uniqueID                 :: IDSupply → ID
leftSupply, rightSupply :: IDSupply → IDSupply
```

The operation `initSupply` yields an initial identifier set. The operation `uniqueID` yields an identifier from such a set while the operations `leftSupply` and `rightSupply` both yield disjoint subsets without the identifier obtained by `uniqueID` (see [9] for a discussion about implementing these operations.). When synthesizing plural functions, we add an additional argument of type `IDSupply` and use the aforementioned operations on it to provide unique identifiers to every choice. The synthesized set function has to pass the initial identifier supply `initSupply` to the plural function. In the case of `double01`, it looks as follows:

```
double01P :: IDSupply → ST Int
double01P s = doubleP (leftSupply s)
                      (Choice (uniqueID s) (Val 0) (Val 1))

double01S :: Values Int
double01S = fromST (doubleP initSupply)
```

With this modified synthesis, the set function yields the expected result `[0,2]`. Note that this extended scheme is necessary only if some operation involved in the definition of the set function has rules with non-linear right-hand sides, i.e., might duplicate argument expressions. For the sake of readability, we omit this extension in the next section where we present another extension necessary when set functions are nested.

7 Synthesis of Nested Set Functions

So far we considered the synthesis of set functions that occur only at the top-level of functional computations, i.e., which are not nested inside other set functions. The synthesis was based on the translation of functions involved in the definition of a set function into plural functions and extracting all the values represented by a search tree into a list structure. If set functions are nested, the situation becomes more complicated since one has to define the plural function of an inner set function. Moreover, choices and failures produced by different set functions, i.e., levels of encapsulations, must be distinguished according to [10]. Although nested set functions are seldom used, a complete implementation of set functions must consider them. Therefore, we discuss in this section how we can extend the scheme proposed so far to accommodate nested set functions.

The original proposal of set functions [4] emphasized the idea to distinguish non-determinism of arguments from non-determinism of the function definition. However, the influence of failing computations and the combination of nested set functions was not specified. These aspects are discussed in [10] where a denotational semantics for functional logic programs with weak encapsulation is proposed. Roughly speaking, an encapsulation level is attached to failures and choices. These levels are taken into account when value sets are extracted from a nested non-determinism structure to ensure that failures and choices are encapsulated by the function they belong to and not any other. We can model this semantics by extending the structure of search trees as follows:

```
data ST a = Val a | Uneval a | Fail Int | Choice Int (ST a) (ST a)
```

The additional argument of the constructors `Fail` and `Choice` specifies the encapsulation level.

Consider the definition

`notf = not`$_{\mathcal{S}}$ `failed` (8)

and the expression `notf`$_{\mathcal{S}}$. Although the right-hand side of `notf` fails because the argument of `not` is demanded w.r.t. the definition (7), the source of the failure is inside its definition so that the failure is encapsulated and the result of `notf`$_{\mathcal{S}}$ is the empty set. However, if we define

`nots x = not`$_{\mathcal{S}}$ `x` (9)

and evaluate `nots`$_{\mathcal{S}}$ `failed`, the computation fails since the failure comes from outside and is not encapsulated. These issues are discussed in [10] where it has been argued that failures outside encapsulated search should lead to a failure instead of an empty set only if there are no other results. For instance, the expression `anyOf`$_{\mathcal{S}}$ `failed` has no value (since the demanded argument is an outside failure) whereas the value of the expression

`anyOf`$_{\mathcal{S}}$ `[failed,1]` (10)

is the set with the single element `1`. This semantics can be implemented by comparing the levels of failures occurring in search trees (see [10] for details).

With the extension of search trees introduced above, we are well prepared to implement this semantics in Curry itself except for one point: outside failures always lead to a failure of the complete evaluation if their value is needed in the encapsulated search. Thus, the evaluation of (10) will always fail. In order to avoid this, we have to transform such a failure into the search tree element `Fail 0` (where `0` is the “top” encapsulation level, i.e., outside any set function). For this purpose, we modify the definitions of `applyST` and `nfST` on arguments matching the `Uneval` constructor by checking whether the evaluation of the argument to a head-normal form fails:[10]

```
applyST f (Uneval x) = if isFail x then Fail 0 else f x
```

```
nfST (Uneval x) = if isFail x then Fail 0 else x `seq` nf x
```

Then one can synthesize plural and set functions similarly to the already presented scheme. In order to set the correct encapsulation level in `Fail` and `Choice` constructors, every function has the current encapsulation level as an additional argument. Finally, one also has to synthesize plural functions of set functions if they are used inside other set functions. For instance, the set function of `not` has type

```
notS :: Bool → Values Bool
```

but the plural function of this set function must represent the result set again as a search tree, i.e., it has the type

```
notSP :: Int → ST Bool → ST (STList Bool)
```

(the first argument is the encapsulation level). To evaluate the search tree structure returned by such plural set functions, we need an operation which behaves similarly to `stValues` but returns a search tree representation of the list of values, i.e., this operation has the type

```
stValuesP :: NF a => Int → ST a → ST (STList a)
```

Note that this operation also takes the encapsulation level as its first argument. For instance, failures are only encapsulated (into an empty list) if they are on the same level, i.e., there is the following defining rule for `stValuesP`:

```
stValuesP e (Fail n) = if n==e then Val Nil else Fail n
```

Choices are treated in a similar way where failures in different alternatives are merged to their maximum level according to the semantics of [10], e.g.,

```
stValuesP 1 (Choice 1 (Fail 0) (Fail 1))
```

evaluates to `Val Nil` (representing the empty set of values).

Now we can define `notSP` by evaluating the result of `notP` with `stValuesP` where the current encapsulation level is increased:

```
notSP e x = stValuesP (e+1) (notP (e+1) x)
```

The plural function of `notf` (8) is straightforward (note that the level of the generated failure is the current encapsulation level):

```
notfP :: Int → ST (STList Bool)
notfP e = notSP e (Fail e)
```

The set function of `notf` is synthesized as presented before except that we additionally provide `1` as the initial encapsulation level (this is also the level encapsulated by `fromST`):

```
notfS :: Values (Values Bool)
notfS = fromST (notfP 1)
```

As we have seen, nested set functions can be synthesized with a scheme similar to simple set functions. In order to correctly model the semantics of [10], an en-

[10] This requires a specific primitive `isFail` to catch failures, which is usually supported in Curry implementations to handle exceptions.

capsulation level is added to each translated operation which is used to generate the correct `Fail` and `Choice` constructors. In order integrate the synthesized set functions into standard Curry programs, arguments passed to synthesized set functions must be checked for failures when their values are demanded.

The extensions presented in the previous and this section can be combined without problems. Concrete examples for this combination and more examples for the synthesis techniques presented in this paper are available on-line.[11] In particular, there are also examples for the synthesis of higher-order functions, which we omitted in this paper due to the additional complexity of synthesizing the plural functions of higher-order arguments.

8 Related Work

The problems caused by integrating encapsulated search in functional logic programs are discussed in [8] where the concepts of strong and weak encapsulation are distinguished. Weak encapsulation fits better to declarative programming since the results do not depend on the order of evaluation. Set functions [4] make the boundaries between different sources of non-determinism clear. The semantical difficulties caused by nesting set functions are discussed in [10] where a denotational semantics for set functions is presented.

The implementation of backtracking and non-determinism in functional languages has a long tradition [25]. While earlier approaches concentrated on embedding Prolog-like constructs in functional languages (e.g., [18, 24]), the implementation of demand-driven non-determinism, which is the core of contemporary functional logic languages [2], has been less explored. A monadic implementation of the call-time choice is developed in [13] which is the basis to translate a subset of Curry to Haskell [7]. Due to performance problems with this generic approach, KiCS2, another compiler from Curry to Haskell, is proposed in [9]. Currently, KiCS2 is the only system implementing encapsulated search and set functions according to [10], but the detailed implementation is complex and, thus, difficult to maintain. This fact partially motivated the development of the approach described in this paper.

9 Conclusions

We have presented a technique to synthesize the definition of a set function of any function defined in a Curry program. This is useful to add set functions and encapsulated search to any Curry system so that an explicit handling of set functions in the run-time system is not necessary. Thanks to our method, one can add a better (i.e., less strict) implementation of set functions to the Prolog-based Curry implementation PAKCS or simplify the run-time system of the Haskell-based Curry implementation KiCS2.

[11] `https://github.com/finnteegen/synthesizing-set-functions`

A disadvantage of our approach is that it increases the size of the transformed program due to the addition of the synthesized code. Considering the fact that the majority of application code is deterministic and not involved in set functions, the increased code size is acceptable. Nevertheless, it is an interesting topic for future work to evaluate the increase of code size for application programs and try to find better synthesis principles (e.g., for specific classes of operations) which produce less additional code.

Our work has the potential of both immediate and far reaching paybacks. We offer a set-based definition of set functions simpler and more immediate than previous ones. We offer a notion of plural function that is original and natural. In Appendix A, we show interesting relationships between the two that allow us to better understand, reason about and compute with these concepts. The immediate consequence is an implementation of set functions competitive with previous proposals.

It is well known that a non-deterministic function can be implemented by a deterministic function that enumerates its results. A direct approach [25] may sacrifice laziness. An approach that preserves laziness [13] is cumbersome for the programmer and may sacrifice efficiency. An intriguing aspect of our work is the possibility of replacing any non-deterministic function, f, in a program with its set function, which is deterministic, by enumerating all the results of f. Our long-term goal is to execute this transformation automatically at compile time without affecting the laziness of a program and without intervention of the programmer. Thus, an (often non-deterministic) functional logic program would become a deterministic program. A consequence of this change is that the techniques for the implementation of non-determinism, such as backtracking, bubbling and pull-tabbing, which are the output of a tremendous intellectual effort of the last few decades, would no longer be an explicit element of the implementation or a concern of the programmer.

References

1. A. Alqaddoumi, S. Antoy, S. Fischer, and F. Reck. The pull-tab transformation. In *Proc. of the Third International Workshop on Graph Computation Models*, pages 127–132. Enschede, The Netherlands, 2010. Available at http://gcm-events.org/gcm2010/pages/gcm2010-preproceedings.pdf.
2. S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *Journal of the ACM*, 47(4):776–822, 2000.
3. S. Antoy and M. Hanus. 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.
4. 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.
5. S. Antoy and M. Hanus. Functional logic programming. *Communications of the ACM*, 53(4):74–85, 2010.
6. R. Atkey and Johann P. Interleaving data and effects. *Journal of Functional Programming*, 25, 2015.

7. B. Braßel, S. Fischer, M. Hanus, and F. Reck. Transforming functional logic programs into monadic functional programs. In *Proc. of the 19th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2010)*, pages 30–47. Springer LNCS 6559, 2011.
8. B. Braßel, M. Hanus, and F. Huch. Encapsulating non-determinism in functional logic computations. *Journal of Functional and Logic Programming*, 2004(6), 2004.
9. B. Braßel, M. Hanus, B. Peemöller, and F. Reck. KiCS2: A new compiler from Curry to Haskell. In *Proc. of the 20th Int. Workshop on Functional and (Constraint) Logic Programming (WFLP 2011)*, pages 1–18. Springer LNCS 6816, 2011.
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. J. de Dios Castro and F.J. López-Fraguas. Extra variables can be eliminated from functional logic programs. *Electronic Notes in Theoretical Computer Science*, 188:3–19, 2007.
12. E. W. Dijkstra. A note on two problems in connexion with graphs. *Numer. Math.*, 1(1):269–271, 1959.
13. S. Fischer, O. Kiselyov, and C. Shan. Purely functional lazy nondeterministic programming. *Journal of Functional programming*, 21(4&5):413–465, 2011.
14. 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.
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/`, 2018.
17. M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.9.0). Available at `http://www.curry-language.org`, 2016.
18. R. Hinze. Prolog's control constructs in a functional setting - axioms and implementation. *International Journal of Foundations of Computer Science*, 12(2):125–170, 2001.
19. H. Hussmann. Nondeterministic algebraic specifications and nonconfluent term rewriting. *Journal of Logic Programming*, 12:237–255, 1992.
20. F.J. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. A simple rewrite notion for call-time choice semantics. In *Proceedings of the 9th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'07)*, pages 197–208. ACM Press, 2007.
21. J.J. Moreno-Navarro, H. Kuchen, R. Loogen, and M. Rodríguez-Artalejo. Lazy narrowing in a graph machine. In *Proc. Second International Conference on Algebraic and Logic Programming*, pages 298–317. Springer LNCS 463, 1990.
22. S. Peyton Jones, editor. *Haskell 98 Language and Libraries—The Revised Report.* Cambridge University Press, 2003.
23. A. Riesco and J. Rodríguez-Hortalá. Singular and plural functions for functional logic programming. *Theory and Practice of Logic Programming*, 14(1):65–116, 2014.
24. S. Seres, M. Spivey, and T. Hoare. Algebra of logic programming. In *Proc. ICLP'99*, pages 184–199. MIT Press, 1999.

25. P. Wadler. How to replace failure by a list of successes. In *Functional Programming and Computer Architecture*, pages 113–128. Springer LNCS 201, 1985.

A Properties of Plural Functions

This section contains some interesting properties of plural functions.

Lemma 1. *The plural function of a function is unique.*

Proof. Suppose that both f_1 and f_2 are plural functions of some function f. Let X be any set such that $f_1\ X = Y_1$ and $f_2\ X = Y_2$, for some Y_1 and Y_2. We show $Y_1 \subseteq Y_2$. For any $y \in Y_1$, by Def. 2, point (1), applied to f_1, there exists some x in X such that $f\ x = y$. Since $x \in X$ and f_2 is a plural function of f, Def. 2, point (2), implies that y is in Y_2. By symmetry, $Y_2 \subseteq Y_1$. Hence, $f_1 = f_2$. □

Lemma 2. *If f and g are composable functions, then $(f \circ g)_{\mathcal{P}} = f_{\mathcal{P}} \circ g_{\mathcal{P}}$.*

Proof. First, we prove that for any X, $(f \circ g)_{\mathcal{P}}\ X \supseteq (f_{\mathcal{P}} \circ g_{\mathcal{P}})\ X$. Suppose $f_{\mathcal{P}}\ (g_{\mathcal{P}}\ X) = Z$ for some sets X and Z. There exists a set Y such that $g_{\mathcal{P}}\ X = Y$ and $f_{\mathcal{P}}\ Y = Z$. If z is some element of Z, then there exists some y in Y such that z is a value of $g\ y$, and there exists some x in X such that y is a value $f\ x$. Consequently z is a value of $(f \circ g)(x)$ and z is an element of $(f \circ g)_{\mathcal{P}}\ X$. The proof that for any X, $(f \circ g)_{\mathcal{P}}\ X \subseteq (f_{\mathcal{P}} \circ g_{\mathcal{P}})\ X$ is similar. □

The following claim establishes key relationships between the set and the plural functions of a function.

Theorem 1. *For any function f, argument x of f, and argument X of $f_{\mathcal{P}}$:*

1. $f_{\mathcal{S}}\ x = f_{\mathcal{P}}\ \{x\}$ *and*
2. $f_{\mathcal{P}}\ X = \uplus\, f_{\mathcal{S}}\ x,\ \ \forall x \in X$.

Proof. We prove that $f_{\mathcal{S}}\ x \subseteq f_{\mathcal{P}}\ \{x\}$. If $y \in f_{\mathcal{S}}\ x$, then, by Def. 1, y is a value of $f\ x$, then, by Def. 2, y is an element of $f_{\mathcal{P}}\ \{x\}$. The proof that $f_{\mathcal{S}}\ x \supseteq f_{\mathcal{P}}\ \{x\}$ is similar. Hence condition (1) holds.
We now prove that $f_{\mathcal{P}}\ X \subseteq \uplus\, f_{\mathcal{S}}\ x$, $\forall x \in X$. For any X, if $y \in f_{\mathcal{P}}\ X$, by Def. 2, there exists some $x \in X$ such that y is a value of $f\ x$. By Def. 1, $y \in f_{\mathcal{S}}\ x$. The proof that $f_{\mathcal{P}}\ X \supseteq \uplus\, f_{\mathcal{S}}\ x$, $\forall x \in X$ is similar. Hence condition (2) holds. □