Synthesizing Set Functions: A Prototype Implementation

Research Project Report

Niels Bunkenburg
Programming Languages and Compiler Construction
Department of Computer Science
Kiel University

Advised by
Prof. Dr. Michael Hanus
M. Sc. Finn Teegen

October 17, 2018

Contents

1. Introduction

When presented with a choice, we usually get to experience only one outcome of it. Whether the decision is about what to eat for lunch, where to live or what career to pursue, we have to decide and can only wonder what would have happened if we picked differently. Would not it be nice if we could collect all outcomes of a decision and pick the most appealing one?

While it is hard to imagine that we someday might be able to pick our lunch based on the recommendation of alternate reality versions of ourselves, the concept is far more tangible in the context of functional logic programming languages. A nondeterministic calculation like `0?1` can yield several values, `0` and `1` in this case. Maybe we would like to count the results or check if a specific outcome is possible? A nondeterministic expression in Curry does not "look" different than a deterministic one, at least in regard to their types. Based on this, checking if `0?1` can be evaluated to a certain result yields a naive implementation.

```
isOne :: Int -> Bool
isOne x = x == 1

check :: Bool
check = isOne (0?1)
```

The intention behind this code is that we would like to know if a value of `0?1` equals one. Unfortunately this is not what the program does. Instead, the function `check` becomes nondeterministic too and yields `True` and `False` because `isOne` is evaluated for every possible value of `0?1`. To express what we originally intended, we need to encapsulate the nondeterminism that might occur in the argument of `isOne`. This is the purpose of *set functions*. A set function returns all values of a nondeterministic expression as a set that can be manipulated like a normal data structure.

```
evalOne :: Int -> Bool
evalOne x = valueOf 1 (set0 x)
```

Curry systems like KiCS2 or PAKCS implement set functions in the respective target language, that is, tree-like structures in Haskell and `findall` in Prolog. This entails slight differences when it comes to handling failures and nonterminating evaluations. Ideally, set functions would be implemented on a higher level, that is, directly in Curry. A recent publication by Antoy et al. [2018] tackles this problem and presents a technique to synthesize the set function f_S of a given f. Based on this, the process of building a working prototype is documented in this report. We begin with the general approach

and a rough overview of the development process, followed by a more detailed look at the design choices and the implementation. The report ist concluded by a final chapter about the limitations, results and future improvements.

2. Approach

The research project is divided into two major parts: Familiarizing with the general topic of set functions by means of a prototype version of "KiCS2.5" and the implementation of a prototype based on Synthesizing Set Functions.

2.1. Time line

The project lasted from April to October 2018. Below is a short overview of what happened in each month.

- April marked the start of the project and began with articles by Antoy and Hanus [2009] as well as Christiansen et al. [2013]. The former describes the semantics of set functions by means of a graph rewriting system while the latter defines an operational semantics. The papers represented a rather abstract introduction of the topic and were supplemented by a first look at the KiCS2 set function implementation.

- May started with a discussion about an issue[1] of KiCS2's set function implementation. The idea of finding and fixing this problem was abandoned due to the complexity of the existing implementation. Instead, working on a minimal implemtation[2] of KiCS2 seemed like a good way to gain a deeper understanding of set functions. The addition of set functions was based upon the implementation presented by Brassel [2010].

- June was the first month with some code output[3]. Along the way of adapting Braßel's implementation, functions to pull up choices to the root of an expression and a normal form transformation were needed. To test the new implementation, the existing examples[4] would have been a good choice but it became apparent that due to the minimalism of the prototype, this would have been a difficult task for the more complex examples. Mid-month the exam preparation period started and the project was paused.

- In July nothing noteworthy – in regard to the project – happened.

[1] https://git.ps.informatik.uni-kiel.de/curry/kics2/issues/15
[2] https://git.informatik.uni-kiel.de/fte/proto_kics
[3] https://git.informatik.uni-kiel.de/stu114713/proto_kics/
[4] https://git.ps.informatik.uni-kiel.de/curry/kics2/blob/master/testsuite/LibraryTests/testSetFunctions.curry

- August marked the end of the first part of the project. The KiCS2.5 set function implementation worked for a simple example but had problems with the duplication of cover information which is used in Braßel's implementation to distinguish different levels of encapsulation. Since the final version of Synthesizing Set Functions was published, the second half of the project began. The goal was to implement a tool that transforms a function within a FlatCurry program into an AbstractCurry program that contains the synthesized set function based on the top-level sharing library. The month ended with the implementation of the plural function transformation.

- September yielded the implementation of totalizing partial functions and the generation of normal form instances. Due to a misunderstanding, the instance generation generated FlatCurry expressions, which unnecessarily complicated the code due to the lack of type classes. Ultimately, the experience proved to be helpful in the implementation of multi-parameter type class instances. To make the code more readable, the transformation was adapted to run within the `StateT` monad. It was discovered that KiCS2 cannot handle the monad instance required to define `StateT`. Therefore, PAKCS was used for the remainder of the project.

- October represented the last part of the project. After implementing the multi-parameter class instances for `toValST` and `FromValST` as well as `toST` and `FromST`, some issues regarding generated function names and the selection of appropriate instance arguments for polymorphic instances were fixed. Finally, all parts of the puzzle were pieced together and resulted in the first prototype of the set function synthesizer. As a last step, call-time choice was implemented.

The first part of the project was meant as preparation for the second part. Therefore, we will not look into the details of the KiCS2.5 implementation and focus on synthesizing set functions. In the next chapter, we explore the design decisions and implementation of the tool.

3. Implementation

The set function synthesizing tool is supplied the name of the module and function name that the set function should be synthesized for. As a result, the necessary plural functions, instances and the set function are returned as pretty-printed AbstractCurry code. Internally, the source code file is transformed into a FlatCurry program and then further processed. In conclusion, the transformation's input is FlatCurry and its output is AbstractCurry code. This low-to-high approach, in terms of language complexity, has two advantages. One, FlatCurry has a multitude of useful program transformation libraries and functions have only one rule by design, which is an assumption made in the paper, too. Two, AbstractCurry supports type classes, which saves us some of the effort to generate type class instances manually. Unfortunately, multi-parameter type classes are not yet implemented in Curry, so some manual instance handling is inevitable.

Over the course of the development process it became apparent that a state is necessary to create unique variable names and to keep the program readable.

3.1. State

The state has a FlatCurry and AbstractCurry program. The former is the initial program while the latter holds the generated code. There are five different maps: two type maps that map type names to generated ST type names and vice versa, as well as one map that maps constructor names to constructors of the ST type and a function map that maps function names to their plural representations. The variable `supplyVarMap` is used in the implementation of call-time choice and stores variables that need to be assigned an `ID` or `IDSupply` value. More on that can be found in subsection 3.2.4. The last elements of the state are a list of type pairs, more precisely the types of a function and its plural function, and a variable that represents the biggest variable index used so far.

```
data State = State
{
    currentProg  :: Prog,
    currentCProg :: CurryProg,
    currentModule :: String,
    currentFunction :: QName,
    typeMap      :: QMap,
    typeSTMap    :: QMap,
    consMap      :: QMap,
    funcMap      :: QMap,
    supplyVarMap :: SMap,
```

```
    funcTypes    :: TList,
    maxVar       :: VarIndex
}

type QMap = FM QName QName

type SMap = FM QName [(VarKind, CTVarIName)]

type TList = [(CTypeExpr, CTypeExpr)]
```

In combination with the Haskell state transformer monad, the resulting code is much more readable than manually folding a state argument through the whole transformation and allows for a demand-driven definition of the operations because demanded functions, types or instances can be looked up and generated if necessary.

3.2. Synthesizing phases

The synthesizing process is divided into multiple phases, which we will explore in this chapter. To make the process easier to follow, the function `anyOf` is used as an example of the different phases.

```
anyOf :: [Int] -> Int
anyOf (x:xs) = x ? anyOf xs
```

3.2.1. CompactFlatCurry

The first step of the transformation is to collect all functions and data types that are used within the transformation target by means of the function `compactFlatCurry` defined in `FlatCurry.Compact`. Since the transformation is mostly demand-driven, this step could be omitted in the future if handling imported modules is implemented.

```
anyOf :: [Int] -> Int
anyOf (x:xs) = x ? anyOf xs

(?) :: a -> a -> a
x ? _ = x
_ ? x = x

data List a = Nil | Cons a (List a)
```

The choice operator appears in the rule of `anyOf` and the list data type in the function type. Therefore, both are added to the program. Since `Int` does not need to be transformed, it is omitted here.

3.2.2. Totalizing partial functions

The definition of `anyOf` is partial because no rule covers the empty list. Since the transformation assumes total functions, the function is totalized in this step. This works by creating a list of constructor name and type declaration pairs for the program. Then, the transformation recursively descends down the function declaration until a case expression is found. Here, the names of covered constructors are collected and the constructors' type is looked up. If a constructor from the type declaration is not covered, a new branch with the expression `failed` is added.

```
anyOf :: [Int] -> Int
anyOf xs = case xs of
  (y:ys) -> x ? anyOf xs
  []     -> failed
```

3.2.3. Lifting case expressions

The transformation requires uniform functions, so nested case expressions can be problematic. Fortunately, the module `LiftCases` already implements lifting nested case expressions, that is, creating new function declarations for nested case expressions. Unfortunately, the module only works with type-annotated FlatCurry because the lifted function's type is generated from the type annotations. To annotate a FlatCurry program with types, the function `inferProg` from the module `FlatCurry.Annotated.TypeInference` is useful. Considering that the type inference does not work well with type classes and that the inferred information is discarded in the next step, it is desirable to implement this step without the usage of annotated FlatCurry.

3.2.4. Plural function transformation

The plural function transformation unifies multiple steps. In general, a FlatCurry function declaration is transformed into an AbstractCurry plural function. To do this, the function name of the transformation target is looked up. A new name for the plural function is generated and an entry in the function map is added to replace recursive calls to the function. Then, the function's type is transformed into an ST type. This is done by adding an `ST` constructor to every top-level argument and then transforming the argument. The interesting part happens when a type constructor is found: If the type lookup yields an ST type, it replaces the previous type. If this is not the case, the type declaration is looked up in the program and the type declaration is transformed.

The type transformation distinguishes two cases. Basic types like `Int`, where all constructors are nullary, and complex types. The basic case is simply dealt with because no transformation is necessary, only an entry like `Int` $\longleftrightarrow$ `Int` is added to the type maps. Complex types get a new ST name that is added to the type maps. Then, the constructors are transformed by adding their ST names to the constructor map and transforming their types.

Finally, the rule of the function needs to be transformed. This is where we need to think about call-time choice for the first time[1]. The plural function has an `IDSupply` argument that we need to distribute between all occurrences of plural function calls and `Choice` constructors that require an `ID`. The approach chosen here is to insert a variable wherever an `IDSupply` or `ID` is needed, combined with an entry in the `supplyVarMap`. The map differentiates both kinds of variables and is used when the function's expression has been transformed. Then, the list of variables in the map are transformed into a list of local declarations where, beginning with the function's `IDSupply` argument, the supply is split into two new supplies where one part is assigned to a variable and the other part is used to create new supplies.

The expression transformation works like described in the paper. Literals are wrapped in a `Val` constructor and variables remain untouched. The `failed` constructor is replaced with the ST constructor `Fail`. Other constructors are replaced with the ST type equivalent if necessary. When a function is found, its name is looked up in the respective map and, if not already done, the function's plural function is generated. The transformation of the other expressions consists mostly of recursive calls of the expression transformation and the adaptation of differences between FlatCurry and AbstractCurry.

```
anyOfP :: IDSupply -> ST (STList Int) -> ST Int
anyOfP v6 v_1 = applyST
  (\v0 -> case v0 of
    STCons v_2 v_3 -> choiceP v5 v_2 (anyOfP v4 v_3)
    STNil          -> Fail
  )
  v_1
 where
   v5 = rightSupply v6
   v7 = leftSupply v6
   v4 = rightSupply v7

choiceP :: IDSupply -> ST t0 -> ST t0 -> ST t0
choiceP v2 v_1 v_2 = Choice v1 v_1 v_2
  where v1 = uniqueID v2

data STList t0 = STNil | STCons (ST t0) (ST (STList t0))
```

3.2.5. Normal form instances

The generation of normal form instances is centered around the function `nfSTCase`, which yields an expression and takes a constructor name, a list of variables that occur in the constructor's pattern and a list of variables that have been bound by case variable patterns. The top-level case expression of `nf` distinguishes the constructors of the ST type. If the constructor does not have arguments, the result is the same constructor

[1]The transformed type has an additional `IDSupply` argument that was not mentioned above.

wrapped in `Val`. Otherwise `nfSTCase` is used to create the nested case expressions. The recursive call of `nf` is applied to the first variable in the constructor variable list and the result is distinguished between `Choice`, `Fail` and the other constructors. The last case is the most interesting one because the result is matched by a variable that the next call of `nfSTCase` receives. The constructor variable that was used in the `nf` expression before is removed from the list and the next nested case expression is created. This procedure is done until the list of constructor variables is empty. Then, the constructor is applied to the list of bound variables that were collected in the calls to `nfSTCase` and wrapped with a `Val`.

```
instance NF t0 => NF (STList t0) where
  nf v9 =
    case v9 of
      STNil -> Val STNil
      STCons v10 v11 ->
        case nfST v10 of
          Choice v12 v13 v14 -> Choice v12 (nf (STCons v13 v11))
                                           (nf (STCons v14 v11))
          Fail -> Fail
          v15 ->
            case nfST v11 of
              Choice v16 v17 v18 -> Choice v16 (nf (STCons v15 v17))
                                               (nf (STCons v15 v18))
              Fail -> Fail
              v19 -> Val (STCons v15 v19)
```

3.2.6. ConvertST instances

Generating the ConvertST requires that we know which types need to be converted. When plural functions are generated, their type is added to the state together with the original function's state. From this information, the necessary instances can be inferred. For example, `anyOf :: [Int] -> Int` and `anyOfP :: ST (STList Int) -> ST Int`[2] are added when transforming `anyOf`. Then, both types are compared by removing the ST constructors and comparing the remaining type constructors. In this case, the result without duplicates is `[([], STList), (Int, Int)]`. Now we know that we need instances for both type pairs. The `(Int, Int)` instance is just a specialized version of `id`.

```
toValST_Int_Int :: Int -> Int
toValST_Int_Int = id

fromValST_Int_Int :: Int -> Int
fromValST_Int_Int = id
```

[2]The `IDSupply` argument is omitted because it does not add any information.

Complex instances like `([], STList)` are generated by iterating over the matching constructor pairs of the types. Since multi-parameter type classes are not supported in Curry yet, type variables result in additional instance function arguments. The rules of the instances are generated according to the pattern shown in the paper. The only difference is that whenever a recursive instance call is necessary, we need to inspect the pattern variable of the constructor for its type. If it is the same type we are currently generation the instance for, a simple recursive call suffices. If it is polymorphic, the appropriate instance function argument (the order of arguments needs to match the order of first occurrence in the type) is chosen. If it is a type that is neither the current instance type nor a polymorphic type, we need to look up the instance in a local map. Since currently only one set function is generated at once, the local map is sufficient because all instances that we generate based on the type comparison are required. When multiple set functions are generated at once, duplicate instances could appear and a global map to look up instances would be more efficient.

```
toValST_List_STList :: (t0 -> ST t1) -> [t0] -> STList t1
toValST_List_STList v20 [] = STNil
toValST_List_STList v21 (v22 : v23) =
  STCons (v21 v22) (toST_List_STList v21 v23)

fromValST_List_STList :: (t0 -> t1) -> STList t0 -> [t1]
fromValST_List_STList v24 STNil = []
fromValST_List_STList v25 (STCons (Val v26) (Val v27)) =
  v25 v26 : fromValST_List_STList v25 v27
```

The remaining `toST` and `fromSt` functions are generated very similarly to the other instances and differ only between basic and complex types in the number of instance function arguments that need to be supplied to the respective `toValST` or `fromValST` function.

```
toST_Int_Int :: Int -> ST Int
toST_Int_Int = Uneval . toValST_Int_Int

fromST_Int_Int :: ST Int -> Values Int
fromST_Int_Int = map fromValST_Int_Int . stValues

toST_List_STList :: (t0 -> ST t1) -> [t0] -> ST (STList t1)
toST_List_STList v28 = Uneval . toValST_List_STList v28

fromST_List_STList :: NF t0 => (t0 -> t1) -> ST (STList t0) -> Values [t1]
fromST_List_STList v29 = map (fromValST_List_STList v29) . stValues
```

3.2.7. Synthesizing set functions

Now that we have created all parts of the puzzle, the only remaining step is to piece them together. The type of the set function is created by wrapping the return type of the

original function in a `Values` constructor. Then, the type is converted to an expression by looking up the correct `fromST` and `toST` instances. Since up to now, polymorphic data type instances have preserved polymorphism, we need to add concrete instance functions based on the set function's type. Polymorphic set functions are not supported. More on the limitations of this implementation can be found in the next chapter.

```
anyOfS :: [Int] -> Values Int
anyOfS v30 =
  fromST_Int_Int (anyOfP initSupply (toST_List_STList toST_Int_Int v30))
```

4. Limitations, Conclusion and Outlook

Due to the prototype nature of the tool, some limitations exist. The current implementation does not support:

- Nested set functions
- Polymorphic set functions
- Higher-order functions
- External functions
- Free variables
- Record types and type synonyms

Nested set functions would require the addition of depth levels to plural functions to distinguish failures as well as the ability to generate plural functions of set functions. While polymorphic set functions could have a theoretical application, they are rare in practice. In contrast, higher-order functions, external functions and type synonyms definitely need to be implemented to have a viable alternative to the current set function implementation.

In conclusion, the prototype shows that the approach presented in the paper can be implemented in a limited setting. When the tool reaches a sufficient degree of language construct coverage and the reliability has been thoroughly tested, integrating it into the Curry preprocessor would require a way to identify set functions in the source code. While the transformation should work for functions of arbitrary arity, it might be useful to keep an adapted version of the current set function interface to allow, for example, better type errors.

Bibliography

S. Antoy, M. Hanus, and F. Teegen. Synthesizing Set Functions. *Proceedings of the 26th International Workshop on Functional and (constraint) Logic Programming (WFLP 2018)*, August 2018.

Sergio Antoy and Michael Hanus. Set functions for functional logic programming. In *Proceedings of the 11th ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, PPDP '09, pages 73–82, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-568-0. doi: 10.1145/1599410.1599420. URL `http://doi.acm.org/10.1145/1599410.1599420`.

Bernd Brassel. Implementing functional logic programs by translation into purely functional programs. 2010.

Jan Christiansen, Michael Hanus, Fabian Reck, and Daniel Seidel. A semantics for weakly encapsulated search in functional logic programs. In *PPDP*, 2013.

A. Repository

The project repository can be found at the following link.

`https://git.ps.informatik.uni-kiel.de/stu114713/2018-setfunctions/tree/master/CODE/setfunctions`

Installing the tool requires a recent Curry installation with the Curry Package Manager. Due to a bug in KiCS2, only PAKCS is supported at the moment. Executing `cypm install` in the project directory installs the executable `synsf`. To use the tool, the name of a Curry module and a function name need to be supplied, for example, `synsf Examples -f anyOf` for the following program.

```
module Examples where

anyOf :: [Int] -> Int
anyOf (x:xs) = x ? anyOf xs
```

The output contains all necessary instances and the generated plural function as well as the synthesized set function. Adding this code to the original file and importing the appropriate ST module (`TopLevelSharing` for the current version of the tool) allows testing the synthesized set function.