Christian-Albrechts-Universität zu Kiel

# An API Search Engine for Curry

**Bachelor Thesis**
Programming Languages and Compiler Construction
Prof. Dr. Michael Hanus
Department for Computer Science
Christian-Albrechts-Universität zu Kiel

**Sandra Dylus**

Advised by M.o.Sc. Björn Peemöller

Sandra Dylus
Matrikelnummer: 1083
Gutenbergstraße 3
24116 Kiel

# Contents

**1 Introduction** **1**
1.1 Motivation . . . . . 1
1.2 Structure . . . . . 1

**2 Preliminaries** **2**
2.1 The programming language Curry . . . . . 2
2.1.1 Functional features . . . . . 3
2.1.2 Logical features . . . . . 4
2.2 Currydoc . . . . . 5
2.3 The Holumbus Framework . . . . . 6

**3 Analysis** **7**
3.1 Indexer . . . . . 7
3.2 Searching . . . . . 8
3.2.1 Parser . . . . . 9

**4 Implementation** **10**
4.1 CurryDoc extension . . . . . 10
4.2 Indexing . . . . . 11
4.3 Parsing user queries . . . . . 11

**5 Conclusion** **14**
5.1 Summary and Results . . . . . 14
5.2 Outlook . . . . . 14

# List of Figures

# List of Tables

# 1 Introduction

## 1.1 Motivation

## 1.2 Structure

# 2
# Preliminaries

This chapter will give a brief introduction to background information that are necessary to comprehend the following chapters. The first section gives an introduction to the programming language Curry. It outlines main concepts and features of the language and gives short explanations for a better understanding. The Curry implementation we refer to in the following sections is PAKCS[1]. Furthermore we present CurryDoc[2], a tool to generate documentation that is distributed with PAKCS, in the second section. The last section introduces the Holumbus[3] framework, a library written in Haskell to configure and build search engines.

## 2.1 The programming language Curry

Curry is a functional logic programming language, that is an international development project to provide a platform for research and teaching mostly. As the description suggest, it offers features of both programming paradigms. This section will start with some general features, followed by two subsections that cover functional and logical concepts of Curry.

Like in Haskell, a program consists of function definitions and data structures. A Curry module *Test* is a program that is saved as *Test.curry*. The syntax of a program is quite similar to Haskell, where function application are also written in juxtaposition.

$$addTwo\ x = x + 2$$

The left-hand side of this function $addTwo\ x$ is evaluated to the right-hand side $x + 2$, i.e. the call $addTwo$ 3 yields $3 + 3 = 5$. Curry supports function definition with pattern matching, which is often used in functional and logical programming languages. That means that instantiated values like *True* and *False* can stand on the left-hand side of a definition. The boolean operation *not* is a good example for a definition with pattern matching, since the result depends on the input. The definition distinguishes between more than one value , so we have to write one rule for each possible input value.

$\neg :: \mathit{Bool} \to \mathit{Bool}$
$\neg\ \mathit{False} = \mathit{True}$
$\neg\ \mathit{True} = \mathit{False}$

The first line indicates that the function expects a boolean value as argument and yields to a boolean value as well. The following two lines are rules, that describe that ¬ *False* yields *True*, whereas ¬ *True* yields *False*. There are no more possible values for the argument of the function ¬, since Curry is a strongly-typed language and *True* and *False* are the only possible values with boolean type. In addition Curry also allows polymorphism. You can use polymorphism for functions that work independent of the value's type. For instance the identity function just returns the argument that you apply to the function. This means you can apply the function to all types of values, because the type does not matter. You use a type variable in your type signature to indicate a polymorphic type. The following code presents the type signature of the identity function.

$\mathit{id} :: a \to a$
$\mathit{id}\ \mathit{value} = \mathit{value}$

Overall a strong type system as well as polymorphism and pattern matching to define functions are among Curry's general features. The next section covers functional characteristics of Curry.

### 2.1.1 Functional features

The boolean values mentioned in the previous section are part of the predefined data structures.

**data** *Bool* = *True* | *False*

This code defines a data structure with the name *Bool* that has to constructors *False* and *True* with the type *Bool*. In this case the constructors are unary. Let's define a data structure that is more interesting and will be used in the following chapters. For example a data structure that represents a Curry program, because we want to create an API search engine for Curry. You can save a Curry program as a module and it exists of functions and types (data structures).

**data** *CurryInfo* = *CurryInfo ModuleInfo* [*FunctionInfo*] [*TypeInfo*]

In this example *CurryInfo* is the name of the type and also the name of the constructor. As a constructor it needs three arguments, a *ModuleInfo* and a list of *FunctionInfo* and *TypeInfo*. We define these data structures that hold the information about specific proporties of the program in chapter 3.

In Curry functions are first-class citizens. This means that they can appear as argument of an expression or even in a data structure. The most popular use-case is the manipulation of all elements of a list by a given function. PAKCS provides the function map, which also exists in Haskell, that takes two arguments, a function and a list and returns a list. The important thing is, that the type of the function matches the elements of the list. For a example a function, that converts an integer to a character can be applied to a list of integers and yields to list of characters. The following code presents the a definition of the function map.

$$\begin{array}{l} \mathit{map} :: (a \to b) \to [\,a\,] \to [\,b\,] \\ \mathit{map\ function}\ [\,] = [\,] \\ \mathit{map\ function}\ (\mathit{element} : \mathit{list}) = \mathit{function\ element} : \mathit{map\ list} \end{array}$$

The syntax for lists is the same as in Haskell. $[\,]$ is the empty list, whereas $1{:}2{:}3{:}[\,]$ is the same as $[1, 2, 3]$ or $1 : [2, 3]$. The first representation is often used in pattern matching, to use the first element of the list in the right-hand side of the definition. The first line presents the type signature. The function $(a \to b)$ takes a value of an unspecific type and returns another unspecific type. The second argument is a list of elements with the type, that the function expects. Furthermore the resulting list contains elements that have the same type as the resulting type of the function. The definition of map says that an empty list yields an empty list, whereas the function is applied to the elements of a list with at least one element recursively.

The last functional concept Curry supports is *lazy evaluation.* In general an expression is evaluated by replacing the left-hand side of a definition by the right-hand side. The evaluation proceeds one replacement after another until it yields a value. A value is an expression that only consists of built-in data structures or literals. If the last replacement does not result in a value, the evaluation fails. If an evaluation has more than one possible replacement step, so called subexpression can be evaluated. Lazy evaluation means that such a subexpression is only evaluated, if its result is necessary to continue the evaluation.

In summary Curry's functional programming features covers data structures, higher-order functions and lazy evaluation as evaluation strategy. Since the logical characteristics of Curry induce that there is another strategy besides lazy evaluation, we will go one to the next section and take a closer look into the logical features.

### 2.1.2 Logical features

Besides the already mentioned functional characteristics, Curry also offers non-deterministic functions and logical variables. Logic programming languages consist of rules, for example a constant function that represents my favourite number.

$$favouriteNumber\ Sandra = 7$$

Let's now assume that I don't have one, but two favourite numbers. Curry as a hybrid of functional and logic programming language allows multiple rules for function definitions.

$$\begin{aligned}&favouriteNumber\ Sandra = 3\\&favouriteNumber\ Sandra = 7\end{aligned}$$

This function is non-deterministic because it returns different values for the same input. The pattern of this function overlaps in functional programming languages, but Curry's ability to search for results allows to define those non-deterministic functions.

Curry also offers logical variables. A variable is called logical, if it appears on the right-hand side but not on the left-hand side of a rule. Free variables are unbound values, that are instantiated to evaluate an expression. It is possible, that it exists more than one binding, since Curry computes all possible solutions of an expression.

Curry provides two different approaches to evaluate an expression with logical variables. The first approach suspends the evaluation in hopes that the logical variable will be bound due to another evaluation of an expression. If there is no other expression to bind the value, the evaluation fails.This approach is called residuation and Curry uses it for boolean operators like the boolean equality $==$. The second approach, called narrowing, guesses a value for an unbound value. Constraint operators like the boolean constraint $= . =$ use narrowing for evaluation. In this context Curry distinguishes two types of operators: flexible operators that narrow and rigid operators that use residuation. For example arithmetic (i.e. $+$, $-$, $*$ etc) and other primitive operations are rigid. However these distinctions do not have any significance for expressions without logical variables, so called *ground expressions*. As mentioned in the previous section, Curry evaluates ground expressions with lazy strategy.

## 2.2 Currydoc

CurryDoc is a tool to generate documentation for a program written in Curry. The current version can generate either a HTML or LaTeX file as output. CurryDoc works similar to code generating tools like javadoc[4] as it uses the comments in source code, which are provided by the user, to gain information about the function or data structure. It also provides the type signatures of functions, since Curry uses a type inferencer algorithm. In addition the CurryDoc tool analyzes the program's structure and approximates the run-time behavior to gain further information[5]. This includes information about in-/completeness, overlapping pattern matches and non-/deterministic functions.

Since CurryDoc is implemented in Curry, it uses the meta-programming module *Flat*[6] that provides an intermediate language of the Curry program to analyze the special function proporties. Such a FlatCurry[7] program consists mainly of a list of functions, a list of types and information of the module itself.

## 2.3 The Holumbus Framework

FH Wedel created this framework in connection with three master theses.

Framework for creating search engines, with indexer and crawler components.

# 3 Analysis

Give a short summary about the following sections.

What do we need to create this search engine? The possibility to generate and update the index.
The possibility to search for these informations in the index. At best: user-friendly

What do we want above all? A web application for queries.

## 3.1 Indexer

What do we need to create an index that can be used for the Curry search engine?

First: Curry specific information (Currydoc)

First start with the idea of the extension: instead of generating a document markup language, generate a readable data structure.

Introduce the TypeExpr data structure that is part of the FlatCurry feature.

```
data TypeExpr =
   TVar TVarIndex                  -- type variable
   | FuncType TypeExpr TypeExpr    -- function type t1->t2
   | TCons QName [TypeExpr]        -- type constructor application
                                   -- TCons (module, name) typeargs
type QName = (String, String)
type TVarIndex = Int
```

Second: a data structure to hold the information (index and document structure from the Holumbus framework)

- HolDocuments - Stores the documents that correspond to the index. A mapping is provided.
- HolIndex - Data structure to store the information, that is traversed in the search process.
- HolumbusState a - the combination of index and document, polymorph by the data the HolDocuments holds.

## 3.2 Searching

How do we search for the information in the index?
Holumbus provides search mechanism with a special syntax.

```
data Query =
    Word String |
    Phrase String |
    CaseWord String |
    CasePhrase String |
    FuzzyWord String |
    Specifier [Context] Query |
    Negation Query |
    BinQuery BinOp Query Query
data BinOp = And | Or | But
```

And this data structure can be processed by processQuery (Holumbus.Query.Processor).

Holumbus also provides a data structure that is returned after a query

```
data Result a   = Result
                  {docHits :: (DocHits a)
                  , wordHits :: WordHits
                  } deriving (Eq, Show)
data DocInfo a = DocInfo
                  {document :: (Document a)
                  , docScore :: Score
                  } deriving (Eq, Show)
data WordInfo = WordInfo
                  {terms :: Terms
```

```
                    , wordScore :: Score
                    } deriving (Eq, Show)
type DocHits a        = DocIdMap (DocInfo a, DocContextHits)
type DocContextHits = Map Context DocWordHits
type DocWordHits      = Map Word Positions
type WordHits         = Map Word (WordInfo, WordContextHits)
type WordContextHits = Map Context WordDocHits
type WordDocHits      = Occurrences
type Score            = Float
type Terms            = [String]
```

But first the user input has to be parsed into the query structure to start the processing.

### 3.2.1 Parser

Which criteria do we want to search for? Modules, functions, types, signatures, and det./non-det., flexible/rigid functions.

First describe the idea, that the use of a specific language increases the usability. But it also restricts the user in her usage of the search engine, if this language gets more complex. So this results in a compromise between a simply to use language and a language that can be parsed. Show the example of searching IO, where the restriction to modules minimizes the result.

Set the focus on signatures. Because Hayoo! does not find signatures with redundant parentheses, Curr(y)gle supports parenthesized signatures and parenthesized query parts in general.

In addition: binary operations/conjunctions.
Explain that in most cases, a combination of more search words is desirable, because first popular search engines like Google$^{TM}$ use this feature so it's common knowledge (the user expects this features) and second it's easier to search for more search words, if the desired result is still vague.
The parser becomes a complex, but very important matter.

# 4
# Implementation

Mention that the implementation is done in Haskell.

Give a short summary of the next sections.

## 4.1 CurryDoc extension

Present the general structure of the *CurryInfo* data and the sub-structures *ModuleInfo*, *FunctionInfo* and *TypeInfo*.

**data** $\mathit{CurryInfo} = \mathit{CurryInfo}\ \mathit{ModuleInfo}\ [\mathit{FunctionInfo}]\ [\mathit{TypeInfo}]$

The name, author and description is the interesting information that is exported.

**data** $\mathit{ModuleInfo} = \mathit{ModuleInfo}\ \mathit{String}\ \mathit{String}\ \mathit{String}\ [\mathit{String}]\ \mathit{String}$

The model contains the name, signature, module, and description of a function and extra information about the non-/determinism and flexible/rigid status.

**data** $\mathit{FunctionInfo} = \mathit{FunctionInfo}\ \mathit{String}\ (\mathit{QName}, \mathit{TypeExpr})\ \mathit{String}$
$\mathit{String}\ \mathit{Bool}\ \mathit{FlexRigidResult}$

The TypeInfo includes the name, signature, list of type variables, module, and description.

**data** $\mathit{TypeInfo} = \mathit{TypeInfo}\ \mathit{String}\ [(\mathit{QName}, [\mathit{TypeExpr}])]\ [\mathit{TVarIndex}]$
$\mathit{String}\ \mathit{String}$

Explain how this extension is used in the CurryDoc tool.
The same data structure is used on the Haskell side that implements the search engine.

## 4.2 Indexing

First mention that the interessting parts (that we want to add to the index) are already filtered by the CurryDoc part. So the indexer only processes the information to the structure provided by the Holumbus framework (HolumbusState).

Present the output (files) the indexer produces. After that introduce the concept of documents and index. In addition to that, say something about refreshing and checking the list of modules.

Mention the three different kinds of documents for the three structures: module, function, type.

```
  -- | Pair of index and documents of the type ModuleInfo
type CurryModIndexerState  = HolumbusState ModuleInfo
  -- | Pair of index and documents of the type FunctionInfo
type CurryFctIndexerState  = HolumbusState FunctionInfo
  -- | Pair of index and documents of the type TypeInfo
type CurryTypeIndexerState = HolumbusState TypeInfo
```

Explain how each kind of information (description, module name, function signature etc) is combined with its context, and that these are stored in the index. Note that information can be extracted by the context again.

Note the difficulties of updating the index, because the data structure of the loaded pair of index and document differs from HolumbusState a. Conversion of *Inverted* to *CompactInverted* and *Documents a* to *SmallDocuments a*.

```
type LoadedIndexerState a = (CompactInverted, SmallDocuments a)
```

Refer to the appendix, where the usage of the curryIndexer is explained.

## 4.3 Parsing user queries

Explain the general idea of parsers: a parser is used to analyze the user query with these different kinds of information. While parsing the string, a new data structure is composed for further use. This data structure is provided by the Holumbus framework and is used to starte the processs that returns the search results.

Introduce the parser type that is parametrized with the type to parse and the resulting type.

```
type Parser s a = [s] → [(a, [s])]
```

Explain the operator (< | >) that applies two parsers and concats the parsing results.

```
(<|>) :: Parser s a → Parser s a → Parser s a
p <|> q = (λts → p ts ++ q ts)
```

Introduce the combination of two parsers that is similar to a monadic bind operator.

```
(<*>) :: Parser s (a → b) → Parser s a → Parser s b
p <*> q = λts → [(f x, ts2) | (f, ts1) ← p ts, (x, ts2) ← q ts1]
```

Now show that a function has to be applied, to use the binding operator.

```
(<$>) :: (a → b) → Parser s a → Parser s b
f <$> p = λts → [(f x, ts1) | (x, ts1) ← p ts]
```

Present a simple example that can parse a given character.

```
parsePredicate :: (Char → Bool) → Parser Char String
parsePredicate predicate (t : ts)
  | predicate t = [([], ts)]
  | otherwise = [([], t : ts)]
parsePredicate _ [] = []
```

Present the concept by writing a parser for expressions with or without parentheses using <|> and <*>.

```
  -- Parses an opening parenthesis by using the predicate parser.
parseOpenParenthesis :: Parser Char String
parseOpenParenthesis =
  parsePredicate ('(' ==)
  -- Parses an closing parenthesis.
parseCloseParenthesis :: Parser Char String
parseCloseParenthesis =
  parsePredicate (')' ==)
  -- Parses a sequence of alpha numeric characters.
parseExpression :: Parser Char String
parseExpression (t : ts) =
  if isAlphaNum t then parseExpression ts else [([], t : ts)]
parseExpression [] = []
  -- Parses an expression or a parenthesized expression.
parenthesizedExpression :: Parser Char String
parenthesizedExpression =
  ((\_ expr _ → expr)
     <$> parseOpenParenthesis <*> parseExpression <*> parseCloseParenthesis)
   <|> parseExpression
```

Mention that the used type in the implementation correlates to *Parser String TypeExpr* (which is already used for the data exchange) for signatures and *Parser String Query* (which is provided by the Holumbus framework) for the end result.
Give a definition of the language (EBNF(?) / appendix).

$query := expr\ [expr] \mid expr\ bool\ expr \mid (expr)$
$expr := (expr) \mid specifier \mid signature \mid string$
$bool := “AND'' \mid “OR'' \mid “NOT''$
$specifier := “:module''\ [alphaNum] \mid “:signature''\ [signature]$
$\mid “:function''\ [alphaNum] \mid “:flexible'' \mid “:rigid'' \mid “:nondet'' \mid “:det''$
$signature := Upper\ alphaNum \mid function \mid constructor$
$function := signature“--> ''\ signature \mid lower“--> ''\ signature \mid$
$signature“\to ''\ lower$
$constructor := Upper\ alphaNum\ signature \mid Upper\ alphaNum\ lower$

# 5 Conclusion

## 5.1 Summary and Results

## 5.2 Outlook

# Bibliography

[1] http://www.informatik.uni-kiel.de/~pakcs/.

[2] http://www-ps.informatik.uni-kiel.de/currywiki/tools/currydoc.

[3] http://holumbus.fh-wedel.de/trac.

[4] http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html.

[5] http://www.informatik.uni-kiel.de/~mh/papers/WFLP02_Doc.pdf.

[6] http://www.informatik.uni-kiel.de/~pakcs/lib/CDOC/Flat.html.

[7] http://www.informatik.uni-kiel.de/~curry/flat/.