JASPER PAUL SIKORRA

FOREIGN CODE INTEGRATION IN CURRY

BACHELOR THESIS

ADVISED BY
Prof. Dr. Michael Hanus
Dipl.-Inf. Jan Tikovsky

RESEARCH GROUP
Programming Languages and Compiler Construction

Institut für Informatik
Christian-Albrechts-Universität zu Kiel

March 2014

ERKLÄRUNG

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbständig und ohne fremde Hilfe angefertigt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Weiterhin versichere ich, dass diese Arbeit noch nicht als Abschlussarbeit an anderer Stelle vorgelegen hat.

Datum *Unterschrift*

Abstract

The integration of formal languages with high specificity in programming languages with many cases of application is often useful. If libraries of the general-purpose-language are utilized to achieve this goal, the formal language easily loses its conciseness. If the formal language is integrated directly, the detection of errors is often problematic. A possible solution is the translation of the formal language before the compilation.

As a proof of concept, the use of regular expressions and format strings in Curry is allowed. The expressions containing foreign syntax are translated and reinserted in the code as functions of Curry libraries. After that, the code is passed to a regular compiler.

By creating a preprocessor for code integration, the conciseness and the detection of type conflicts can be preserved. As a byproduct a tool is created, which is open to enhancement with other domain specific languages.

CONTENTS

Part I

INTRODUCTION

1 MOTIVATION

This thesis discusses the conception of a platform for the integration of domain specific languages (DSLs) in Curry[5] and its realization.

At the moment domain specific languages may only be integrated if they are modified to use the same syntax and semantics as Curry. This has some advantages: there is no need to alter the compiler and the benefits of a full-grown declarative programming language remain intact. On the downside the domain specific language often loses its conciseness if the syntax is transformed. This loss of clarity

```
-- Regex syntax
(\+|-)?[0-9]+
-- Curry syntax
[Times (0,1) ([Xor ([Literal ('+')]) ([Literal ('-')])
   ]),Plus ([Bracket [Right (('0'),('9'))]])]
```

Listing 1: Regular expression in original and Curry syntax

is displayed paradigmatically using regular expressions in Listing 1.

Another problem surfaces if the domain specific language is embedded using its original syntax instead of libraries of the host language. If this solution is chosen, the detection of errors, especially type conflicts, becomes very difficult without modifying the compiler. This problem is shown exemplarily in Listing 2 in Haskell[1], a func-

```
import Text.Printf

pr :: Int -> IO ()
pr a = printf "%s" a
```

Listing 2: A flawed printf call in Haskell

tional programming language very similar to Curry. The program in the example is flawed, because the argument of **printf** should be a string, but the function **pr** expects an integer. Though erroneous, there is no problem compiling this example with GHC[2]. This is a problem that occurs regularly if domain specific languages, which do not use the syntax of the host language, are used in a general purpose

1 See http://www.haskell.org/onlinereport/intro.html for further information on Haskell.
2 The Glasgow Haskell Compiler, https://www.haskell.org/ghc/

language. If the use of arbitrary data types in regular expressions is allowed, similar issues arise. This implies that the integration of domain-specific formal languages is often problematic in respect of type safety.

In this thesis a system for code integration is examined that tackles both problems. By allowing the use of domain specific languages in their native syntax the conciseness is kept, by translating the expressions, which differ from Curry syntax, into functions of pure Curry, strict typing and other advantages of the host language apply.

As a proof of concept, the use of two domain specific languages, regular expressions and format strings with their unique syntax is allowed. Regular expressions provide a method to describe formal languages and are a well explored field of theoretical informatics. Format strings are used for the formatting and insertion of variables in strings. The syntax and semantics of format strings are simple and they are still used on daily basis in different programming languages, making them a great candidate for prototyping a platform for embedded domain specific languages.

For the distinction and definition of domain specific languages in Curry, a syntactic construct is specified, which allow the detection

```
import Format
printOut name age = ``format "hello %s. your age %i",
   name,age''
```

Listing 3: Curry with an integrated format string expression

of foreign code. The recognized expression are translated by language specific parsers and reinserted in the source code, resulting in pure Curry programs. This method of translation is basically a preprocessor for a Curry compiler. It enables the use of any computer language with its own syntax, as long as a translator exists that is able to convert the computer language to legal Curry code. Since Curry is a Turing complete language, any computer program could be integrated. This feature is particularly useful for domain specific languages, though.

This thesis is segmented in four major parts.

In Part ii the fundamentals needed to integrate code are discussed: the functional-logic programming language Curry is introduced, the concept behind domain specific languages and their embedding is explained and the basic functions of translators are examined.

In Part iii the implementation of the Code Integrator is discussed, followed by a description of the libaries and parsers for format strings as well as regular expressions.

In Part iv the measurement of the run time complexity of the Code Integrator is delineated briefly.

Concluding the thesis, the results are analyzed and possible prospects are described in Part v.

Part II

FUNDAMENTALS

2 CURRY

2.1 FUNCTIONAL-LOGIC PROGRAMMING

Curry is a declarative, general purpose programming language. The objective of declarative programming is to describe the problem and its solutions instead of the method to solve it.[4]

A distinguishing mark of Curry is the amalgamation of three sub-paradigms of declarative programming. The concepts of functional, logic and constraint programming are interwoven in Curry to create a new paradigm: functional-logic programming.

Functional programming is based on the λ-calculus, which was introduced in the 1930s by Church.[13] The main goal of functional programming is the minimization of side effects, which is achieved by restricting the languages to evaluation of functions. Popular functional programming languages are Haskell and a dialect of Lisp called Scheme[1].

Logic programming tries to utilize mathematical logic in programs. Logic programs mostly consists of facts and rules and permit the use of partial information. A major logic programming language is Prolog[2].

Constraint programming is a generalization of the logic programming paradigm. It programming extends logic programming with evaluation under arithmetic and finite domains.[6] The enhancement of Prolog with constraint satisfaction is one example.[3]

Often problems are solved easier under one paradigm than under another. With Curry it is possible to use the advantages of all three concepts in one language. To accomplish this, different ideas are taken from all three paradigms. Functional programming characteristics like higher order functions and lazy evaluation are imported, logic programming supplies the possibility to work with partial information and logic variables, constraint programming delivers the option to use constrained variables. By linking those paradigms new paths in problem solving derive.[4]

2.2 PROGRAMS IN CURRY

The syntax and semantics of Curry are specified in the Curry Report[5]. A Curry program contains only type and function declarations and

1 See `http://www.r6rs.org/final/html/r6rs/r6rs-Z-H-1.html` for further information on Scheme.
2 A free implementation of Prolog is located at `http://www.swi-prolog.org/`.
3 A description of a constraint programming library for Prolog can be found at `http://www.swi-prolog.org/man/clpfd.html`.
4 The later introduced functional-logic Parser Combinator is a good example.

module description necessary for modularization. The syntax of Curry is very similar to Haskell.

2.2.1 *Data types*

Data types in Curry are declared with a type constructors of the form

$$\text{data } T\ \alpha_1 \dots\ \alpha_n = C_1\ \tau_{11} \dots\ \tau_{1n_1} \mid \dots \mid\ C_k\ \tau_{k1} \dots\ \tau_{kn_k}$$

T is called the type constructor, $C_1, \dots, C_k$ data constructors and $\alpha_1, \dots, \alpha_n$ type variables. $\tau_{11}, \dots, \tau_{kn_k}$are types again. In this way, data types can be defined recursively, meaning that T can reappear as a type τ on the right side. The declaration also determines the type of the constructor C_i, which is

$$\tau_{i1} \to \dots \to \tau_{in_i} \to T\ \alpha_1 \dots\ \alpha_2$$

This implies that data constructors can be conceived as functions, which accords with the ideal of functional programming that everything should be a function. In Listing 4 **Maybe** is declared**,** a poly-

```
data Maybe a = Just a | Nothing
```

Listing 4: The data type Maybe

morphic data type with one type variable and two data constructors. The first data constructor **Just** awaits one parameter of the type **a** , the other data constructor **Nothing** does not have a parameter. Some other declarations of data types can be found in Listing 5. **List**

```
data Bool = True | False
data List a = [] | a : List a
data Tree a = Branch (Tree a) (Tree a) | Leaf a
```

Listing 5: More data types

and **Tree** are defined recursively, the data constructor of **Bool** has no parameters. In Curry the data types for Booleans (**Bool**), constraints (**Success**), functions, integers (**Int**), floats (**Float**), lists, characters (**Char**), strings (**String**) and tuples are predefined.

If a data constructor $C_i\ \tau_{i1} \dots\ \tau_{in_i}$ is applied in the form $c_i\ t_{i1} \dots\ t_{in_i}$ then $t_{i1} \dots\ t_{in_i}$ has the type $\tau_{i1} \dots\ \tau_{in_i}$. In Listing 6 the data type **Tree** is used to define an actual binary tree.

For easier comprehension of the code, Curry enables the declaration of type synonyms. Those have the form

```
tree :: Tree Int
tree = Branch (Leaf 1) (Leaf 4)
```

Listing 6: Application of a data constructor

$$\text{type } T\ \alpha_1 \dots\ \alpha_n = \tau$$

where T is the new type constructor and τ is the synonymous type. Synonym types are interchangeable anywhere in the code.

2.2.2 *Functions*

Functions are defined in Curry in the form

$$f\ t_1\ \dots\ t_n = e$$

f is the functions name, e the functions body and $t_1\ \dots\ t_n$ are data terms, meaning that they are either variables or data constructors, which again may contain data terms. The corresponding types to functions may be declared explicitly in the form

$$f : \tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow \tau$$

A function type declared like this assigns the data terms $t_1, \dots, t_n$ the types $\tau_1, \dots, \tau_n$ and e the type τ . In Listing 7 a square function is

```
square :: Int -> Int
square x = x * x
```

Listing 7: Power of two function

defined on integers. In this case the explicit type of the square function is redundant because the operator $*$ is only defined on integers, meaning the same function type would be determined by type inference.

Functions may also be written as conditioned equations in the form

$$f\ t_1 \dots\ t_2 \mid c = e$$

These equations are only evaluated if the condition c is satisfied.

Curry supports pattern matching and multiple occurrence of the same data term on the left side, which is disabled in Haskell. These kind of functions are evaluated if the condition that the data terms

```
absolute x | x < 0      = (-1) * x
           | otherwise = x
```

Listing 8: Function with conditions

are equivalent is satisfied. Another feature that is not part of Haskell is the possibility of using non-deterministic functions in Curry. Non-deterministic functions consist of two or more equations which are evaluated under the same conditions. In this way one function can yield different results with the same input. In Listing 9 the function

```
throwCoin = "head"
throwCoin = "tail"
```

Listing 9: Non-deterministic functions

throwCoin is declared, which will yield two values on a call: **"head"** and **"tail".** Curry also supports local definitions, which are functions, variables or data terms limited to a defined domain. Local definitions are defined in Curry with the keywords **let ... in** and **where.**

```
powerOfFour x = let y = square x
                in square y
```

Listing 10: Function with a local definition

2.2.3 *Free variables*

Free variables must be declared locally in Curry with the keywords **where** ... **free**. For each of those variables, which may also be called

```
f x | x =:= y = y
        where y free
-- is equal to
f x = x
```

Listing 11: Function with a free variable

unbound, values are computed which fulfill the equation. This is done using a technique called narrowing. If we consider an example call of f in Listing 11 with $x = 3$ then a possible solution for the constraint is computed ($y = 3$) and after that the function is evaluated ($f\ 3 = 3$).

2.2.4 *Comments*

Comments in Curry are designated in the same way as in Haskell by using **{- ... -}** or **- -** ... and are ignored by the compiler.

2.2.5 *Layout rule*

Another similarity to Haskell is the layout rule used in Curry. Indentation may be used instead of curly brackets and semicolons to identify syntactic contexts.

This means if lines following a keyword should belong to the same syntactic entity, they need greater indentation than the keyword. The minimal indentation is set by first term following the keyword. All other lines in the same context must have the same or greater indentation.

```
f x = h x where {g y = y+1 ; h z = (g z) * 2 }
-- equals
f x = h x
  where g y = y+1
        h z = (g z) * 2
```

Listing 12: Layout rule example

Exemplarily the layout rule is depicted in Listing 12. The local function **h** belongs to the same syntactic entity as **g** and **where**.

2.2.6 *Type system*

Typing is the mapping of terms on types. If a term is assigned a type, the term can only assume values in the range described by the type. Type conflicts occur if a term has to assume a value, that is outside of the range set by the type. Since type conflicts can be found at compile time, typing is mostly used to detect errors in programs at an early state.

In Curry every variable, data constructor and function has a static type. The types of data constructors have to be declared explicitly, the types of variables and functions may be stated or ascertained by a type inference system. Curry uses the type inference proposed by Hindley/Milner, which allows polymorphic types.

Polymorphic types contain type variables, which are able to assume arbitrary types. The data type **Maybe**, depicted in Listing 4, is a polymorphic type with one type variable. In Listing 13 the **map** and the **addThree** functions are defined. The type of **addThree** is not declared, but the Curry compiler still computes it.

```
map :: (a -> b) -> [a] -> [b]
map _ []      = []
map f (e:es) = (f e):(map f es)

addThree l = map ((+) 3) l
-- Infered type (:t addThree)
-- addThree :: [Int] -> [Int]
```

Listing 13: Type inference

Types in Curry are defined at compile time which allows the detection of type conflicts at a very early stage. For example if a function awaits an integer but is passed a string, the compiler is able to recognize the type conflict and will yield an error.

3 DOMAIN SPECIFIC LANGUAGES

3.1 INTRODUCTION

Generic and specific approaches are discerned in many fields of science.[14] Generic approaches apply to a wide, specific approaches to a small set of problems. In computer science two kinds of programming languages are distinguished by this characteristic: general purpose languages (GPLs) and domain-specific languages (DSLs). Historically many languages that are counted as general purpose languages today were created to perform in a certain area of computation.[14] By growing into general purpose languages the need of creating specialized problem solvers reemerged. Two different solutions were developed to handle this task: subroutine libraries and domain specific languages.

- **Subroutine libraries** are the classic method for solving a particular problem in a general purpose language.[14] They supply reusable functions which are tailored for domains and use the syntax and semantics of the GPL. An example subroutine library is the Parser module supplied with Curry which is fitted to the particular assignment of parsing strings.

- **Domain specific languages** are usually small languages with a narrowed, sometimes unique syntax and high expressiveness aligned to solve a limited domain of problems.[3]

Today, there is a great diversity of DSLs separated in different classes like programming, markup and modeling. Domain specific programming languages sometimes contain general purpose languages and thereby provide expressive power for a group of problems, while still being able to handle others. On the other hand there are languages like YACC[1], which focus on generating only components or libraries for bigger applications.

A typical domain specific programming language is SQL whose only purpose is the creation and manipulation of relational databases. In the example depicted in Listing 14, the exceptional syntax which is adjusted to this particular aim becomes obvious. Another domain specific language, which is used on wide base, is the markup language HTML. This language is suited only for one task: the creation of web pages presentable by a web browser.

1 YACC (abbreviation for "Yet Another Compiler Compiler") is a LALR parser generator. See `http://www.csa.syr.edu/~chapin/cis657/yacc.pdf` for further information.

```
SELECT Lecture.title, Professor.name
FROM Professor, Lecture
WHERE Professor.PersNo = Lecture.PersNo
```

Listing 14: An SQL query

```
<!DOCTYPE html>
<html>
        <head><title>This is a title</title></head>
        <body><p>Hello world!</p></body>
</html>
```

Listing 15: An HTML page

The gap to a general purpose programming language is easily recognizable. In this project two domain specific languages were integrated into Curry: regular expressions and format strings.

3.2 REGULAR EXPRESSIONS

The domain specific language of regular expressions (abbreviated to regex) is often used in text processing. It also plays a part in theoretical informatics and formal language theory. Those formal languages, which can be described with regular expressions, are called regular and are the least expressive languages in the Chomsky hierarchy.[11]

3.2.1 *Definition of Regular Expressions*

In its basic form, a regular expression is a string r containing terminal symbols of an alphabet Σ and some meta symbols. A regular expression is then defined by the following rules:

1. $\emptyset$ is a regular expression
2. if $a \in \Sigma$ then a is a regular expression
3. if s and t are regular expressions, then st (Concatenation), $(s|t)$ (Alternative) and $s*$ (Star) are regular expressions

The relating semantics can be found in Table 1. For example, the Star operator specifies the application of the expression any amount of times. The regular expression in Listing 16 recognizes any string containing only x and y and closing with xy. In the POSIX standard for regex more operators are defined which eases the creation of more complex expressions. In the specification regular expressions are divided in Basic Regular Expression (BRE) and Extended Regular Expression (ERE) which use slightly different syntax.

regular expression	recognized language
ε	$\{\varepsilon\}$
$a \in \Sigma$	$\{a\}$
st	$L_s L_t$
$(s\|t)$	$L_s \cup L_t$
$s*$	L_s*

Table 1: Semantics of regular expressions

```
(x|y)*xy
```

Listing 16: A simple regular expression

```
(cats|dogs) are (very )* nice!{1,3}
```

Listing 17: Another regular expression

The example shown in Listing 17 utilizes the ERE syntax, which allows the use of unescaped curly brackets that denote how often a regular expression has to be matched. In this case the exclamation mark has to be matched between one and three times.

3.2.2 *Types in Regular Expressions*

In most cases regular expressions are defined to match strings, which means the terminal symbols are characters. But regular expressions can also be applied on generic lists of any kind, as long as a total order is defined on the type of the elements. If this condition is satisfied, each unique list element can be assigned a terminal symbol and the normal syntax and semantics of regexs can be applied. In this way regular expressions can be defined with polymorphic types.

3.3 FORMAT STRINGS

The language of format strings is a very simple domain specific template language well-known from its use in the printf function in C. A format string consists of normal characters and placeholders called format specifiers, which describe how a data type should be depicted. It is used in similar ways in many programming languages including Lisp, Perl, PHP, Python, Java and Haskell.

```
printf("Color %s","blue"); // Output: Color blue
```

Listing 18: A printf call in C

3.3.1 *Definition of Format Strings*

A format string is composed of common characters and format specifiers and has zero or more corresponding arguments. Each argument is bound to one specifier and their attribution is implied by their order of occurrence. While common characters can simply be copied to the output string, format specifiers describe the formatting of the assigned argument and are replaced by the result. A format specifier is introduced by the character '%' followed by zero or more **flags**, an optional minimum field **width**, an optional **precision** and a conversion specifier setting the type of conversion. Consequently, it is of the form

%[flags][width][.precision]type

where square brackets denote optionality. The type is only one character and determines the necessary type of the associated argument, as well as its kind of conversion. For instance, if the type is 'X', then the argument must be an integer and will be converted to hexadecimal number system. The type may be the character '%'. In this case, the format specifier will only yield this character and will not have an assigned argument.

Width and precision are either integral numbers or the character '*'. In the second case, the format specifier awaits another argument of type integer, which defines the width or precision. The width controls the minimum number of characters with which the format specifier should be replaced. The precision describes the number of characters used for displaying the converted argument. While the width can never truncate or round a result, for the precision this is well possible. The flags describe further modifications to the result, setting the alignment, the use of an algebraic sign and the character to fill whitespaces with.

```
float fl = 3.1468931
printf("%0*.4f", 8, fl); // Output: 003.1469
```

Listing 19: A printf call with a more complex format specifier

3.3.2 *Types in Format Strings*

In ordinary format strings only few types of arguments are possible. Those are integers, floats, strings and characters. In C more types like pointers are allowed, which are irrelevant in Curry.

The correctness of the argument types depends on the format specifiers contained by the string. This implies that the format string must be static at compile time if type safety at that point is desired, and the string must be parsed before or during compilation.

4 INTEGRATION OF DOMAIN SPECIFIC LANGUAGES

4.1 INTRODUCTION

Two standard procedures in implementing a domain specific languages can be discerned

- **Interpretation and Compilation** is the classical approach. The domain specific language is integrated by creating a new compiler. This means there is no need for compromises. Type-checking, error handling and optimization can be done directly on the domain level which leads to high effectiveness.
- **Extension of a base language** will reuse a given language and enhance it to fit the domain. Using this concept, the capabilities of the base language remain intact leading to lesser effort.[9]

The first model is often selected for languages like SQL which benefit hugely from specialized compilers. For small languages like regular expressions and format strings the second way seems to fit better. This approach is attempted with the Code Integrator.

4.2 EXTENSION OF BASE LANGUAGES WITH DSLS

For the extension of base languages with domain specific languages some models have evolved, the most important ones are domain specific libraries, preprocessing and compiler extension.

- The **embedding of a DSL with domain specific libraries** uses the syntactic mechanisms of the base language for the implementation. This procedure requires the least effort, because the compiler of the base language is reused in its original form. On the downside, restricting the syntax often makes the domain specific language lose its biggest benefit: the concision.
- The second model allows arbitrary syntax for the domain specific language and translates the DSL to the base language with a **preprocessor** before compilation. This model often uses domain specific libraries, too, to facilitate the translation procedure. The downside of this approach is the sacrifice of domain level optimization.
- **Extending the compiler** is a similar approach as preprocessing only that domain level optimization becomes possible by having access to different levels of the compile process.

An example for the first approach is the Parser library in Curry discussed in Section 5.2.2. This model is also used by the functional programming language Lisp by allowing macro expanders, which provide possibilities to enhance the Lisp syntax. Template Haskell[1] uses a method, which is similar to the second approach. The third approach is widely used in TCL[14] allowing a huge variety of DSLs.

4.3 CODE INTEGRATION THROUGH PREPROCESSING

A preprocessor is used in computer programs to prepare input for a later state. Programming languages like Lisp and Scheme use preprocessing to transform imperative and object-oriented programs into functional programs and to enable the creation of minilanguages.[2]

The Code Integrator is a preprocessor used to translate code of a domain specific language into a general purpose language like Curry. After the translation, the code is compiled with a standard Curry compiler. The advantage of this method is the retention of the original syntax of the DSL, the possibility to use Curry's safety mechanisms like strict type-checking and the missing necessity to manipulate the compiler. Beside the impossibility of domain level code optimization, another drawback is the limited capability to deliver precise errors. Errors found by the compiler in the translated code are given in the base language and are therefore catchier to fix. In Figure 1 the process of translation and compilation is elucidated. The preprocessor needs its own error handling and errors could be passed to the compiler if it provides this feature. In Listing 20 input and output of the Code

```
import Regex
-- before preprocessing
onlyAs s = s ``regex a*''
-- after preprocessing
onlyAs s = s `match` ([Star ([Literal ('a')])])
```

Listing 20: Regex in Curry before and after Preprocessing

Integrator are depicted. The output program uses the domain specific Regex library.

1 See https://research.microsoft.com/en-us/um/people/simonpj/papers/meta-haskell/meta-haskell.pdf for a descript of Template Haskell
2 See https://www.gnu.org/software/emacs/manual/html_node/elisp/Macros.html for further information on Lisp macros

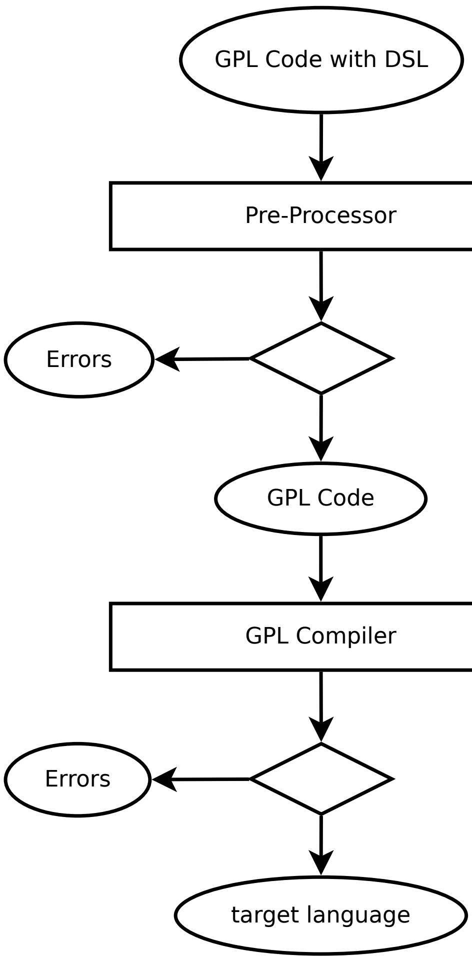

Figure 1: Preprocessing a DSL

5 TRANSLATORS

A translator is a computer program, used to convert one formal language into another. It is often used to translate a higher programming language into machine code.[1] In this thesis a translator is applied to transmute Curry program code with integrated domain specific languages into pure Curry program code. Typically, the procedure of translation is done in six steps:

1. lexical analysis (Scanner)
2. syntactic analysis (Parser)
3. semantic analysis
4. intermediate code generation
5. program optimization
6. code generation

While all steps are important for regular compilers, only the steps 1, 2 and 6 are necessary for the Code Integrator.

5.1 SCANNER

The task of a scanner is the lexical analysis, which means separating the input in a chain of tokens. The resulting decomposition of the input into logical units is an abstraction, which eases the task of parsing on the next level.[10]

Normally, the input of a scanner is a string of characters and the scanner is a finite state automaton accepting a regular grammar. The scanner recognizes the valid atomic components of the languages, which are often keywords like operators and constants. All atomic components are then converted to corresponding tokens. The result is a list of tokens, which constitute the partition of the input strings syntax into the smallest logical units.

5.2 PARSER

A parser recognizes relations between the tokens and puts them in a structure, in this way facilitating the work of the following levels and validating the syntax of the input. The input of a parser is a list of tokens. The parser verifies their use in the right context and, in theory, transfers them into an abstract syntax tree (AST). Often such a structure is never fully built.[10]

Two major parsing algorithms exist, which are used on common base: top-down and bottom-up parsing. Their names describe the way they build the AST. Both algorithms only work if the recognized language is produced by a deterministic context-free grammar.[11]

5.2.1 *Parser-Combinators*

In functional programming languages, Parser combinator libraries are built to simplify the creation of parsers. A parser made of parser combinators consists of many atomic parsers, which are joined together to create a new one. This new parser is able to recognize a more complex grammar. An atomic parser works like a normal parser, expecting a list of tokens and returning an abstract syntax tree, if the parsing was successful.

For example, if two parsers are linked as alternatives, the languages identified by each parser are recognized.

5.2.2 *Functional-Logic Parsers*

By interweaving concepts of functional and logic programming, new kinds of parsers arise. The Curry Parser library is similar to other parser combinators, but uses free variables to detect sequences of characters in a string. The concept of this library and of functional-logic parsers in general was developed by Caballero and Lopez-Fraguas [2] and implemented in Curry by Hanus[1].

A functional-logic parser using the Curry Parser library is shown in Listing 21. This parser recognizes positive integers. Therefore, operations like sequential application <*> or alternatives <||> are provided. With >>> a representation is added to the parser.

```
import Parser
import Char

posInt = digit d <*> digits ds         >>> (d:ds) where
    d,ds free
digits = empty                         >>> []
     <||> digit d <*> digits ds        >>> (d:ds) where
         d,ds free
digit  = satisfy (\c -> isDigit c) ch >>> ch      where
    ch free
```

Listing 21: Parsing a positive integer with the functional-logic parser

1 The implementation of the functional-logic parser can be found at http://www-ps.informatik.uni-kiel.de/kics2/lib/CDOC/Parser.html

5.3 TARGET CODE GENERATION

After parsing, the abstract syntax tree is transformed into code of the target language. Often specific libraries are created to ease this task by providing data types and functions, which reflect the structure of the parsed language.

Part III

IMPLEMENTATION

6 IMPLEMENTATION OF FOREIGN CODE INTEGRATION IN CURRY

6.1 SPECIFICATION

Foreign integrated code is embedded in Curry files. This means outside of it normal Curry syntax rules apply. Those can be found in the Curry Report[5] and are partly described in Section 2.2. Preferably, the parser is ought to recognize as few syntactic constructs of Curry as possible. This simplifies the task and accelerates the process of parsing. Of cause, constructs like quotations and comments need to be identified, because they escape the integrated code from translation.

For the integration, a new syntactical construct is build to allow the identification of foreign code by the parser. This task is not trivial because a huge variety of languages must be recognizable if extensibility should be preserved. To solve this problem, a special syntax for the embedding is specified:

AccentGraves Langtag Whitespaces DSLExpression SingleQuotes

Integrated foreign code starts with two or more accent graves and is terminated by the same amount of single quotes. The minimal amount of accent graves necessary is determined by the occurrence of accent graves and single quotes inside the integrated code. If n accent graves or n single quotes appear in a row, at least $n+1$ accent graves and single quotes must bracket the integrated code. This implies that the grammar of the language of integrated code is neither context-free nor context-sensitive. Therefore, it is of Type-0 of the Chomsky hierarchy.[10]

The accent graves are followed by the language tag. The language tag is a string containing anything but whitespaces, tabs and newlines. At the moment, the following language tags have corresponding translators: *format*, *printf*, *regex*, *html* and *xml*. The language tag is followed by an arbitrary amount of whitespaces, tabs and newlines. The indentation of the first term following the language tag, which is the initial of the foreign code, sets the offset of the residual rest of the expression. In this way, the normal Curry layout rules apply outside of the integrated expression and new layout rules may be used inside the foreign language. Some examples for legal integrated code are depicted in Listing 22.

This specification allows the identification of integrated code and the corresponding foreign language in Curry.

```
initEmail s = s ``regex
  [a-zA-Z0-9]
  ([a-zA-Z0-9\._])*

tailEmail s = s ``regex @
                          [a-zA-Z0-9][a-zA-Z0-9\-]*\.
                          ([:alnum:][a-zA-Z0-9\-]*\.)*
                          [a-zA-Z]{2,4}''

printEl f = ``printf "%+20.3E",f'' >> putStr " km\n"
             >> return ()

regexer s = s ````regex ''''```''''''
```

Listing 22: Legal integrated code

6.2 SYSTEM DESIGN

As mentioned before, the used concept for the Code Integrator is preprocessing. The preprocessor parses Curry code, which may contain deviating designated syntax of a domain specific language. The identified expressions of the domain specific languages are then passed to translators and converted to Curry expressions. The resulting expressions are combined in their original order to generate the target code. This procedure is displayed in Figure 2. It allows the easy extension with more domain specific languages.

6.2.1 *Error Handling*

If the domain specific language code is integrated incorrectly or is flawed, specific errors should be returned. This includes the output of lines and columns where the errors occurred and tailored error messages.

A concept to handle errors in functional language parsers is the use of parser monads. The Happy parser generator [1] utilizes this model to allow easy error definition and handling.

To improve the usefulness of Curry compiler errors with translated domain specific language code, they should point to the right position in the original domain specific language code. This means the starting lines of the code blocks should not be altered. This is achieved by inserting newlines in the translated code.

1 See haskell.org/happy/doc/html/sec-monads.html for a detailed description on how Happy provides error handling.

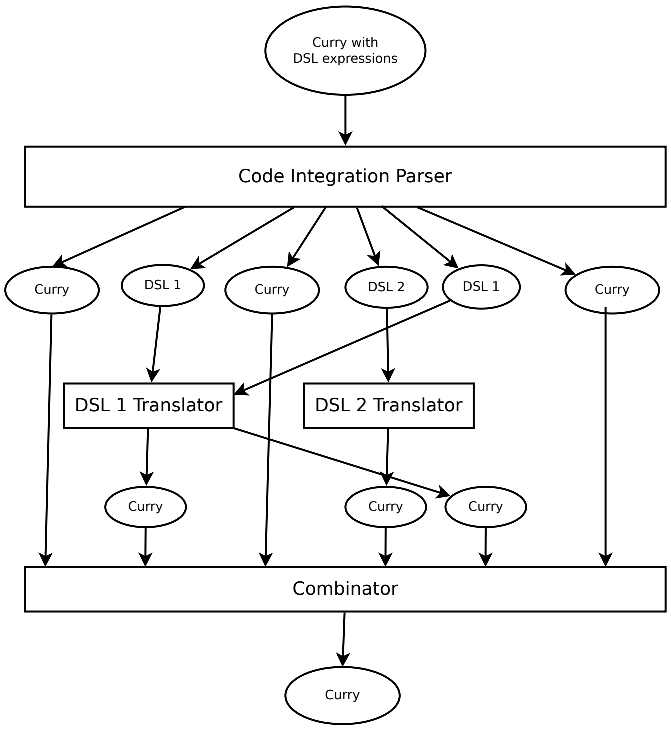

Figure 2: Preprocessing of Curry with integrated code

6.2.2 *Layout Rule*

The offset of the integrated code should confirm to Curry layout rules. This means the Code Integration parser needs to remove the offset in every line of the domain specific language code to expose the real code. Therefore the offset has to be calculated. If the code conflicts

```
printSome a b = ``format
        "%s %d",
                a''
-- expression without the offset
"%s %d",
        a
```

Listing 23: Curry layout rule with integrated code

with the layout rule, specific error messages must be created.

6.2.3 *Domain Specific Libraries*

For each domain specific language, a Curry library and a translator must be created. The translators have to expect a position as well as a string and must return the translated string. They must also pass the errors occurring during the translation to the main module. The creation of regex and format libraries and parsers are described in Chapter 7 and Chapter 8.

6.3 REALIZATION

6.3.1 *The Objective*

The first step in the process of implementation is the creation of a parser that transfers Curry code with integrated expressions into a special data type. This data type, depicted in Listing 24, is used for both, integrated expressions and the Curry code between them, and provides two positions, the language identifier and the expression

```
data StandardToken = StTk Pos Pos (Maybe Langtag) Code
```

Listing 24: Declaration of the StandardToken in ParseTypes.curry

of the domain specific language. The first position is the one of the whole code that is transformed, the second position marks the beginning of the DSL's expression. This is necessary to allow DSL specific translators to create errors with correct lines and columns.

After this parser is built, it can be used in the translator to disassemble the input, which is then passed to the translators and concatenated.

6.3.2 *The Parser Monad*

The parser used for the recognition of the integrated expressions should pass errors and warnings with attached positions back to the main translator. As a consequence, error, warning and position handling have to be available in nearly every function in the Code Integration parser. To ease this task, a library for handling positions and three monads, one for errors, one for warnings and one for combining both, were created.

```
data Pos = Pos Filename Absolute Line Column
initPos :: Filename -> Pos
lnDifference :: Pos -> Pos -> Line
colDifference :: Pos -> Pos -> Column
movePosByChar :: Pos -> Char -> Pos
movePosByString :: Pos -> String -> Pos
```

Listing 25: Parsing Positions data type and some functions

For the task of providing precise positions a library, partly depicted in Listing 25, was created. This library simplifies the task of moving the position by characters or strings and is also used for calculation of offsets in the foreign code.

For the handling of errors and warnings two monads were created. Those two monads were then combined, using a method that was generally described by King/Wadler[8]. The type and the signatures of some functions of the resulting parser monad are shown in Listing 26. An error is thrown by calling *throwPM* with a position and an

```
--- Combining ParseResult and Warnings monads into a
    new monad type
type PM a = WM (PR a)
--- Encapsulate an Error Monad with a Warning Monad
    creating a Parse Monad
warnPM :: PR a -> [Warning] -> PM a
--- Bind
bindPM :: PM a -> (a -> PM b) -> PM b
--- Lift
liftPM :: (a -> b) -> PM a -> PM b
--- Return without Warnings or Errors
cleanPM :: a -> PM a
--- Return without Errors but with Warnings
warnOKPM :: a -> [Warning] -> PM a
--- Throw an Error
throwPM :: Pos -> String -> PM _
--- Remove the Warning Monad from PM
discardWarningsPM :: PM a -> PR a
--- Extract the Warnings
getWarningsPM :: PM a -> [Warning]
--- Apply a function on each Warning
mapWarnsPM :: (Warning -> Warning) -> PM a -> PM a
--- Join multiple Parser Monads into one
sequencePM :: [PM a] -> PM [a]
```

Listing 26: Parser Monad

error message. Warnings are attached to the result using the return function *warnOKPM*. If no warnings and errors occurred, *cleanPM* is used, a function that is the normal unit function of the monad.[7]

6.3.3 *The Parser*

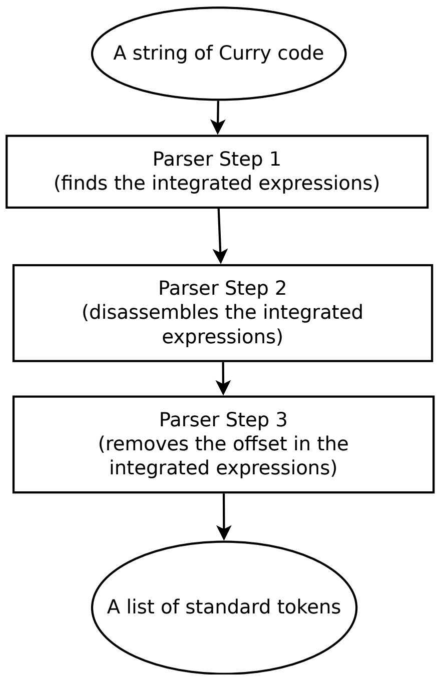
A string of Curry code
Parser Step 1
(finds the integrated expressions)
Parser Step 2
(disassembles the integrated expressions)
Parser Step 3
(removes the offset in the integrated expressions)
A list of standard tokens

Figure 3: Steps in the Code Integration Parser

With the *StandardToken* data type in Listing 24 declared, the objective is obvious: take an arbitrary string containing Curry with or without integrated expressions and return an equivalent list of *StandardToken*s. The implied task was solved by creating a parser with three main steps, which are shown in Figure 3. In the first step the complete string is parsed and quotes, single quotes, comments, integrated expressions as well as normal code are recognized. After that the integrated expressions are decomposed into a language tag and foreign code. This code still holds the offset, which is part of the surrounding Curry code. Therefore, the offset is removed in the next step. The result is a list of *StandardToken*, which may then be used for the specific translators.

The recognition of the integrated expressions in the first step is done using the data type and the function, shown partly in Listing 27 and Listing 28. If a character is found that introduces an expression which needs special handling, a sub-function is called to parse it. Otherwise, the character is cached in an accumulator. In this way, a sequence of characters are kept until a special character is read. Then the sequence is used to create a *Normal* data type. The function used

```
data L1Token = Normal Pos String -- normal Curry
             | Exp    Pos        -- DSL code
                      Int        -- number of idents
                      String     -- DSL code
```

Listing 27: Data type of the first level of the Parser

for the parsing on this level needs to recognize constructs that belong to normal Curry, for example block comments and quotations, too.

The second and third step of the parser are done in one unit on the next level. Here, all *L1Token*s are converted to *StandardToken*s. The integrated expressions are disassembled on this level, separating

```
parserL1Iter acc accP p s@(c:cs)
  -- Parse Quotations
  | c == '\\"' = passThrough parseQuotation (
     movePosByChar p '\\"') cs
  | ...
  | otherwise =
   -- Recognize integrated expressions
     let (n,r) = countAndDrop s_ident s
     in if (min_number <= n)
          -- Parse integrated expressions
          then passThroughInt parseIntegrated
```

Listing 28: The first level of the parser in CIParser.curry

the language tag and the code of the foreign language as well as calculating its offset and the position. The function which is used for this task is shown in Listing 29.

```
disassembleIntExp (Exp p i s) =
    let
        -- Recognize the language tag
        (langtag,rest1) = break isSpace s
        -- Recognize the whitespaces and the dsl
        (spaces,dsl) = span isSpace rest1
    in
      if (null langtag) then throwPM p err_no_langtag
         else
           (...)
           bindPM cleanDSL
             (\cDSL -> cleanPM (StTk p posBeforeDSL (
                Just langtag) cDSL))
```

Listing 29: Function for the disassembling of integrated expressions

First, it decomposes the integrated expression in a language tag, whitespaces and the foreign code. Then the functions calculates the offset by calling the function *movePosByString* on the leading identifiers, the language tag and the following spaces and extracting the resulting column of the position. Subsequently, the offset is removed from the foreign code with help of a sub-function, which is applied on each line. After this step the result is in form of a list of *Standard-Token* and is ready for the specific translators.

6.3.4 *The Translator*

The translator exploits the Code Integration parser to create a list of *StandardToken*s from the input. The list elements are then passed to the corresponding foreign language translators. The returned strings are concatenated and newlines and whitespaces are inserted if necessary. The result may then be written into a file. In Listing 30 the

```
translateString :: String -> String -> IO String
translateString name s =
  do stw <- concatAllIOPM $ applyLangParsers $ ciparser
       name s
     putStr (formatWarnings (getWarnings stw))
     return $ escapePR (discardWarnings stw) errfun
```

Listing 30: The translation function for Strings

translation function for a string containing Curry with integrated expressions is shown. The parameter *name* is a arbitrary identifier, for example a file path, relating to origin of the input string *s*. *s* is parsed with the Code Integration parser and the integrated code is passed to specific translators, depending on the language tag. The results are then concatenated and warnings and errors are shown if necessary.

7 IMPLEMENTATION OF A REGEX LIBRARY AND A REGEX PARSER

7.1 THE REGEX LIBRARY

To allow the use of regular expressions in Curry, a library is created. It should provides a function to match a string with an arbitrary regular expression. For this purpose, the function *match*, depicted in Listing 32, and the data type Regex, depicted in Listing 31, are declared.

```
type Regex a = [ORegex a]
data ORegex a = Nil
              | Literal a
              | Xor (Regex a) (Regex a)
              | Star (Regex a)
              | Plus (Regex a)
              | AnyLiteral
              | Bracket [Either a (a,a)]
              | Times (Int,Int) (Regex a)
                            | ...
```

Listing 31: Regex data type

A Regex is a list of polymorphic Regex operators. For each operator, a specific data constructor is provided. The data constructor *Nil* matches the empty list, *Xor* is the alternative operator and so on. *Xor*, *Star*, *Plus* and *Times* have to contain regular expressions again to fulfill their purpose.

This data type may now be used in the *match* function to check whether a string conforms to its grammar. The *match* function terminates if the regular expression, which is tested for matching a string, is empty. If the string is empty too, the matching is successful, otherwise it isn't. If the Regex is not empty, the head element is matched with part of the string. In some cases this is quite easy, for example for *Nil* or a simple *Literal*, in other cases it requires the use of extra sub-functions. Matching with regular expressions containing the *Star*, *Plus* or *Times* operator is exceptionally tricky, because firstly, for each repetition the whole residual string has to be matched with the residual Regex and secondly, each repetition may match multiple initials of the string.

For the Star operator, the corresponding function solving this task is shown in Listing 33. The *matchstar* function first tries to match the string with the Regex zero times and then creates a list of Boolean

```
match :: [a] -> Regex a -> Bool
match s r = case r of
  []                    -> if (s == []) then True else
     False
  (Nil:ors)             -> match s ors
  (Xor or1 or2:ors)     -> match s (or1 ++ ors) || match
     s (or2 ++ ors)
  (Literal c:ors)       -> case s of
    []       -> False
    (d:ds)   -> if (d == c) then match ds ors else False
  (Star r:ors)          -> matchstar s r ors
  (...)
```

Listing 32: match function

```
matchstar :: [a] -> Regex a -> Regex a -> Bool
matchstar st r rgx = (||)
  (match st rgx)
  (tryeach (map (\x -> match x r) (inits st)) (tails st
     ) r rgx)

tryeach :: [Bool] -> [[a]] -> Regex a -> Regex a ->
    Bool
tryeach [] []          _  _    = False
tryeach (b:bs) (t:ts) r   rgx = (||)
    (if b then (match t rgx || matchstar t r rgx)
          else False)
    (tryeach bs ts r rgx)
```

Listing 33: Matching with the Star operator

values which represents whether an initial of the string is matched with the Regex once. With this list and the corresponding tails of the String the *tryeach* function is called. If the list contains True as head, the matching of the residual string is tried. If this doesn't lead to success, the next initial of the string is tried.

7.2 THE REGEX PARSER

7.2.1 *Specification*

Regular expressions are identified in Integrated Code with the language tag *regex*. Their syntax and semantics conform to the Extended Regular Expressions (ERE) defined in IEEE Std 1003.1. with one exception:[1] in the Code Integrator the operators < and > used to identify variables or Curry data terms. The selection of those identifiers implies that arrow brackets in regular expressions cannot be used in variables or data terms without escaping them. The escaping is done with a leading backslash. Two examples displaying the inclusion of Curry expressions in Regex are depicted in Listing 34.

```
madeOf :: [a] -> a -> a -> Bool
madeOf l v1 v2 = l ``regex (<v1>|<v2>)*''

soTrue :: Bool
soTrue = "a" ``regex <((\c -\> c) 'a')>''
```

Listing 34: Variables in Regex

The possible use of variables and data terms is the only difference to Extended Regular Expressions.

7.2.2 *Parsing*

The regex parser enables the conversion of normal regex to Curry expressions. Therefore, the regular expression must be parsed and translated into the regex data type shown in Listing 31. Exemplarily, the input and output of the translation process are shown in Listing 35. The procedure of the translation may be divided in three

```
-- Before
check s = s ``regex [a-z]*''
-- After
check s = s `match` ([Star ([Bracket [Right (('a'),('z
    '))]])])
```

Listing 35: Conversion of a regex

steps: tokenizing, parsing and generation of the output string. The tokenization of the input string is the easiest step, matching characters and escaped characters on tokens. Only special characters like operators have their unique token, other characters are just transformed into one token containing them.

After that, the parsing of the list of tokens into a data structure is carried out. The structure is displayed in Listing 36 and is very similar to the one used in the Regex library, except the type variable is set to string. This is necessary to enable the use of Curry data terms in the regular expressions.

1 A free draft can be found at http://www.open-std.org/jtc1/sc22/open/n4217.pdf. The same specification is described at http://pubs.opengroup.org/onlinepubs/009695399/basedefs/xbd_chap09.html

```
type Regex = [ORegex]
data ORegex = Nil
            | Literal String
            | Xor Regex Regex
            | Star Regex
            (...)
            | Bracket [Either String (String,String)]
            (...)
```

Listing 36: Data structure after the parsing

```
parse :: Pos -> PM Regex -> [Token] -> PM Regex
parse _ prr []      = prr
parse p prr (t:ts) = bindPM prr (\r ->
  case t of
      TokenStar         -> liftPM ((:) (Star r)) (parsen
          p ts)
      TokenBar          -> parseBar p r ts
      TokenPoint        -> liftPM ((++) r) (parse p (
         cleanPM [AnyLiteral]) ts)
      (...)
      TokenOSBracket    -> liftPM ((++) r) (
         parseOSBracket p ts)
      (...)
      TokenLiteral c    -> liftPM ((++) r)
                                  (parse p (cleanPM [
                                     Literal (show c)])
                                      ts))
```

Listing 37: The Regex Parse function

The main function for parsing is shown in Listing 37. This function expects the position of the regular expression, the regular expression which was parsed before, and a list of tokens and returns a Regex. The necessity to keep the last parsed regular expression derives because ERE use suffix operators. The *parse* function forwards the input to specific sub-functions depending on the head of the tokens.

After the main function returns, the output is processed by a code generation function which yields a string containing data types and functions of the Regex library. This code generation function basically converts the Regex data type of the parser to the one of the library.

8 IMPLEMENTATION OF A FORMAT STRING LIBRARY AND FORMAT STRING PARSER

8.1 THE FORMAT STRING LIBRARY

The Format library allows the conversion of variables to strings and is therefore similar to the show function in Curry. The major difference is the possibility to call the functions of the library with special parameters, which define the formatting of the variable. The treatable data types and the possible formatting operations are implied by the format string expressions specified in Section 3.3. To enable type safety, a function for each type is created. At the moment functions for Strings, Chars, Integers and Floats are implemented. These primitives were chosen because they are the standard types used in format expressions. The translator described in Section 8.2 utilizes the functions of the library in the generated target code.

The formatting of the variable is determined by four data types:[1]

- The *Typ*, not to be mistaken for the actual type in Curry, describes how the variable is represented. For example, an integer may be converted to hexadecimal or octal numeric system.
- The *Flag* holds information about the use of signs, decimal points and alignment.
- The *Width* specifies the minimal amount of characters of the string.
- The *Precision* defines the exact amount of characters to be printed, for example after the decimal point of a float.

The formatting function for strings is depicted in Listing 38. This function ignores the Typ and all flags but the minus flag. The precision and width parameters are applied and then the string is returned.

Other format functions are more complicated. For example, the function for formatting integers needs to convert integers into different numeric systems.

Another complicated part of the implementation is the formatting of floats, occasionally they have to be rounded or the exponent has to be eliminated. To ease those tasks, a new data type for floats is created. Corresponding to the data type, new functions are declared to round and modify floats.

1 A more exact description of those arguments can be found in Section 3.3.

```
type ShowSpec a = Typ -> Maybe Flag -> Maybe Width ->
   Maybe Precision -> a -> String
(...)
showString :: ShowSpec String
showString _ mf mw mp s =
  let flags      = convertFlags mf
      width      = convertWidth mw
      minusFlag  = getMinusFlag flags
      afterPrec  = maybe s (flip take s) mp
      afterWidth = if minusFlag
                     then fillWithCharsLeftAlign width '
                          ' afterPrec
                     else fillWithCharsRightAlign width
                         ' ' afterPrec
  in afterWidth
```

Listing 38: showString function

8.2 THE FORMAT STRING PARSER

In the Format String parser, the functional-logic parser combinator described in Listing 20 is exploited. This reduces the effort of constructing the parser, but is slightly slower because of the unbound variables that need additional time to be evaluated. The syntax of the parser combinator is similar to context free grammar definitions. In Listing 39 one part of the parser is depicted. This part is on the highest level of the parser and utilizes parsers on lower levels. *expression* reflects the structure of a format string, consisting of a quoted string and some variables. The quotes string consists of two quotation marks with common characters and specifiers in between.

```
expression    =  quoted q <*> vars v >>> (q,v) where q,v
    free
quoted        = terminal '\"' <*> strsAndSpecs s <*>
   terminal '\"' >>> s where s free
strsAndSpecs = empty >>> []
  <||> str st         >>> [Left st]
  <||> spec sp <*> strsAndSpecs stsps >>> (Right sp:
      stsps)
  <||> str st <*> spec sp <*> strsAndSpecs stsps >>> (
      Left st:Right sp:stsps) where st,sp,stsps free
```

Listing 39: Part of the Format Parser

This parser is called by a function that eliminates the non-determinism, that might arise from the use of free variables. After this function is applied, the variables are assigned to specifiers. At this point, the data type already looks very similar to the one used in the Format library. The result is then used to generate the target code, which consists of normal strings concatenated with format specifiers in the form of calls to the library.

Part IV

EVALUATION

9 RUN TIME COMPLEXITY

To evaluate the run time complexity of the Code Integrator, big files with repetitive patterns were created. They either

- raise the amount of integrated expressions,
- raise the complexity of the integrated expressions
- or contain pure Curry code.

In all measurements the files were translated using the Code Integration 1.8 compiled with kics2 0.3.1. The time was measured using the UNIX command **time** on a 2.2 GHz dual core notebook with 4 GB of RAM. An example command used in this procedure is depicted in Listing 40. To translate the foreign code, the dummy parser was used,

```
time ./bin/Translator examples/performance/5000dummy.
   incurry e.curry
```

Listing 40: Bash command to measure the time of translation

which only replaces newlines spaces and has linear time complexity.

Files with multiple occurrences of the line depicted in Listing 41 were used for the measurement on the translation of high amounts of integrated expressions. The line was copied 5000, 10000, 15000 and

```
``dummy Hello!''
```

Listing 41: Line to measure huge amounts of integrated expressions

30000 times into files, which were then used as the input of the Code Integrator. Table 2 lists the results. It is obvious, that the translation

Number of Lines	Time
5000	0.410s
10000	0.774s
15000	1.125s
30000	2.311

Table 2: Results on huge amounts of integrated expressions

time is proportional to the number of lines. This is an indicator for linear time complexity.

Sequences	Characters	Time
1	409	0.005s
~50	9913	0.037
~100	19013	0.063s
~200	37831	0.113s

Table 3: Results on more complex integrated expressions

If the difficulty is raised by adding more symbols to the integrated expressions, similar results occur. For example, four files were created with a high amount of inner single quotes and accent graves. The amount of outer accent graves and single quotes were always 100 and multiple sequences of 99 accent graves and 99 single quotes were placed in the integrated expression in the way depicted in Listing 42. In the first file only one sequence was used, which results in about 400 characters. In the second file 50 sequences were used, which results in about 9900 characters, in the third file 100 sequences were used, which results in about 19000 characters and in the fourth file 200 sequences were used, which results in about 37800 characters.

~~~
`````dummy ````''''Hello!'''''
~~~

Listing 42: Line of more complex integrated expressions

Again, the outcome, depicted in Table 3, indicates linear run time complexity.

The Code Integrator was then tested on large files that do not contain integrated expressions. Therefore, the CIParsers.curry source code was inserted multiple times in a file and then translated. This is a reasonable way to measure the performance, because CIParser.curry contains all syntactic constructs of Curry, which could escape integrated expressions, like quotes and comments, but does not hold foreign code. The time was measured with the same method as described above.

CIParsers Inserted	Lines of Code	Time
1	380	0.037s
2	760	0.079s
5	1900	0.148s
10	3800	0.277s
20	7600	0.551s
40	15200	1.170s

Table 4: Results on big, legal Curry files

As depicted in Table 4, the consumed time for the translation process is proportional to the amount of lines in the code. Each of the results indicates that the parser for integrated expressions has linear run time complexity.

Part V

CONCLUSION

10 RESULT

The purpose of the Code Integrator was to provide an extendable platform which allows the integration of domain specific languages in Curry. Different concepts for the incorporation of DSLs in host languages exist, each with up- and downsides.

The creation of domain specific libraries is easily accomplished, but if directly used, reduces the conciseness of the language. If the DSL is embedded using its own syntax and an external compiler, domain level optimization is possible, but safety mechanisms of the host language are bypassed. The modification of an existing compiler to support the DSL is complicated and hard to maintain.

Therefore, the selected approach was the construction of a preprocessor. In this way, the conciseness of the DSL and the safety mechanisms of the host language were kept.

With the Code Integrator, a platform was designed and implemented that enables the use of DSLs with their original syntax in Curry. The foreign code is translated to Curry before the compilation, which allows the exploitation of its static analysis.

In the implementation, the translation is done by parsing specific syntactic constructs, which contain the expressions of the DSL and do not occur in regular Curry. The expressions are then passed to translators which return equivalent legal Curry code.

The implementation was evaluated and all results indicate, that the parse process that recognizes the integrated expressions and decomposes them and the process that recombines the translated code have linear time complexity.

With the selected approach, the type safety of the host language can be kept. For example, if the flawed printf example in Listing 2 is embedded using integrated code (Listing 43), the compilation of the translated file will break because of a type error.

```
import Format

pr :: Int -> IO ()
pr a = ``printf "%s",a''
-- Inferred type: [Prelude.Char] -> Prelude.IO ()
-- Expected type: Prelude.Int -> Prelude.IO ()
```

Listing 43: Flawed integrated printf expression

Regex and Format String were selected to prototype the integration of foreign code in Curry. For both languages, specific libraries were created to provide core functions. After that, parsers for the conversion of the DSLs were implemented.

With this task completed, Regex and Format String may now be used in Curry in the form of integrated code.

The Code Integrator is built to allow the simple addition of more translators. A way to enhance the Code Integrator is described in Appendix. A translator for real HTML to Curry's HTML structure was developed by Deppert[1] and was integrated in the Code Integrator without major troubles.

Code Integration using a preprocessor is a viable option for the enhancement of Curry. The advantages of the Code Integrator are clear: conciseness and static safety mechanisms are kept. A disadvantage is the impossibility to optimize the domain specific language during compilation. The only way to improve the DSL expression is to write better corresponding libraries in Curry and by generating better code with the translator.

All in all, the Code Integrator is a simple, maintainable and extendable solution for the integration of domain specific languages.

1 The Bachelor thesis on this topic by Max Deppert is currently in progress.

11 PROSPECT

The incoporation of the Code Integrator as a built-in preprocessor for a Curry compiler is the main goal of near future work. The compiler could enable the use of the translator, for example if a flag is passed.

Another task for the future is the extension of the Code Integrator with more domain specific languages. The addition of more DSLs is well possible, which was already proved by Deppert. A DSL, which is a good candidate for such an enhancement, is the relational database language SQL. An obstacle for realizing this task might be the vastness of SQL's syntax and the necessity of a corresponding SQL library in Curry.

Possible improvement of the Code Integrator could also arise from the use of parallelization. Since each integrated expression is translated separately, it might be possible to do this simultaneously. The overhead created by parallelization might lead to ineffectiveness, though.

The implementation of a search algorithm in the regular expressions library is a viable task, too. Instead of only matching whole lists, regular expressions could be used to search and replace parts of a list. A search algorithm is described by Thompson[12] and only needs to be transferred to Curry.

APPENDIX

REQUIREMENTS AND INSTALLATION

The compilation of the Code Integrator requires an installed Curry compiler[1], the build-management-tool make[2] and the source code of the project[3]. The right Curry compiler must be set in the Code Integrator's Makefile as CC. Using Windows it might be necessary to modify some more options in the Makefile.

The source code can easily be compiled by running **make** in the main folder.

In Listing 44 an example on how to install the Code Integrator using pakcs and sed is shown. After the compilation the executable

```
# Change directory to the Code Integrators main folder
$ cd /path/to/CodeIntegrator
# set the correct curry compiler binary in the makefile
$ sed -i '48s/.*/CC = kics2/' Makefile
# run make
$ make
```

Listing 44: Example Compilation Process

is placed in the bin folder of the Code Integrator. The Makefile also supports the cleaning of the project folder by running **make clean**.

1 Compilers can be found at http://curry-language.org.
2 The GNU Make reference can be found at https://www.gnu.org/software/make/ and the nmake reference for Windows at http://msdn.microsoft.com/de-de/library/dd9y37ha.aspx
3 The source code is currently hosted nonpublic at https://git-ps.informatik.uni-kiel.de/theses/2013-jsi-made-ba

To translate a file with the Code Integrator execute the Translator binary with two file paths as parameters: the file path to the input file and the file path to were the translated file should be placed. Then use a Curry compiler to compile the resulting file. If libraries outside of the standard Curry search path are used, add them to the compilers search path. Ways to do this can be found in the compiler's manual. Remember to import the correct libraries already in the file that should be translated. For syntax and semantics of integrated

```
# Translate
$ sh path/to/CodeIntegrator/bin/Translator /path/to/
   input/file /path/to/outputfile
# Add libraries to search path
$ CURRYPATH=$CURRYPATH:/path/to/CodeIntegrator/lib
$ export CURRYPATH
# Use with compiler
$ pakcs :l /path/to/outputfile
```

Listing 45: Translating and compiling a file

code in Curry see Section 6.1, the Translator.curry module and the examples.

Real World Example

The input file shown in Listing 46 is saved at $HOME/email.curry.

```
import Regex
isEmail s = s ``regex [a-zA-Z0-9]([a-zA-Z0-9\._])* @[a-
   zA-Z0-9][a-zA-Z0-9\-]*\.([:alnum:][a-zA-Z0-9\-]*\.)
   *[a-zA-Z]{2,4}''
```

Listing 46: Input file

```
$ sh $HOME/Bachelor/CodeIntegrator/bin/Translator $HOME
   /email.curry $HOME/email_trans.curry
$ CURRYPATH=$CURRYPATH:$HOME/Bachelor/CodeIntegrator/
   lib && export CURRYPATH
$ pakcs :l $HOME/email_trans.curry
ex1> isEmail "jsi@informatik.uni-kiel.de"
Result: True
```

Listing 47: Real world example

The enhancement of the Code Integrator with more domain specific languages is well possible. To integrate a new language a translation function with the type

$$\text{Pos} \rightarrow \text{String} \rightarrow \text{IO (PM String)}$$

must be implemented and able to convert the DSL syntax to legal Curry code. The data type **PM** and the relating functions are defined and described in the **ParseMonad.curry** module. The **Pos** data type and functions can be found in **ParsePos.curry**. The new translation function must be imported and added to the **cases** in the **parsers** function in the **Translator.curry** module in the same way as it is shown in Listing 48.

```
parsers :: Maybe Langtag -> LangParser
parsers = maybe iden pars
  where
    iden _ s = return $ cleanPM s
    pars :: Langtag -> LangParser
    pars l p =
      case l of
        "dummy"      -> DummyParser.parse p
        "format"     -> FormatParser.parse p
        "printf"     -> FormatParser.parse p
        (...)
        -- EXAMPLE ENHANCEMENT
        "new_lang"   -> NewLangParser.parse p
        _            -> (\_ -> return $ throwPM p ("Bad
           langtag: " ++ l))
```

Listing 48: parsers function in Translator.curry

BIBLIOGRAPHY

[1] Alfred Vaino Aho and Jeffrey David Ullman. *Principles of compiler design*. Addison-Wesley series in computer science and information processing. Addison-Wesley, Reading (Mass.), 1977. Autre tirage : 1978, 1979. (Cited on page 20.)

[2] Rafael Caballero and Francisco Javier López-Fraguas. A functional-logic perspective on parsing. In Aart Middeldorp and Taisuke Sato, editors, *Fuji International Symposium on Functional and Logic Programming*, volume 1722 of *Lecture Notes in Computer Science*, pages 85–99. Springer, 1999. (Cited on page 21.)

[3] Martin Fowler. *Domain Specific Languages*. Addison-Wesley Professional, 1st edition, 2010. (Cited on page 12.)

[4] M. Hanus. Multi-paradigm declarative languages. In *Proceedings of the International Conference on Logic Programming (ICLP 2007)*, pages 45–75. Springer LNCS 4670, 2007. (Cited on page 6.)

[5] M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.8.3). Available at `http://www.curry-language.org`, 2012. (Cited on pages 2, 6, and 24.)

[6] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, pages 111–119, New York, NY, USA, 1987. ACM. (Cited on page 6.)

[7] Mark P. Jones and Luc Duponcheel. Composing monads. Technical report, 1993. (Cited on page 28.)

[8] David J. King and Philip Wadler. Combining monads. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, pages 134–143, London, UK, UK, 1993. Springer-Verlag. (Cited on page 28.)

[9] Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *Proceedings of the 2Nd Conference on Domain-specific Languages*, DSL '99, pages 109–122, New York, NY, USA, 1999. ACM. (Cited on page 17.)

[10] Thomas Pittman and James Peters. *The Art of Compiler Design*. Prentice-Hall, Englewood Cliffs, NJ, November 1991. (Cited on pages 20 and 24.)

[11] Stefano Crespi Reghizzi. *Formal Languages and Compilation*. Springer Publishing Company, Incorporated, 1 edition, 2009. (Cited on pages 13 and 21.)

[12] Ken Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, June 1968. (Cited on page 46.)

[13] Simon Thompson. *Type theory and functional programming*. International computer science series. Addison-Wesley, 1991. (Cited on page 6.)

[14] Arie Van Deursen, Paul Klint, and Joost Visser. Domain-specific languages. *Centrum voor Wiskunde en Informatika*, 5, 2000. (Cited on pages 12 and 18.)