Benefits of implementing a query language in purely functional style

Artúr Poór, István Bozó, Tamás Kozsik, Gábor Páli, Melinda Tóth

poor_a@inf.elte.hu, bozo_i@inf.elte.hu, kto@inf.elte.hu, pgj@inf.elte.hu, toth_m@inf.elte.hu

ELTE Eötvös Loránd University
Faculty of Informatics
Budapest, Hungary

Abstract

Software maintenance can be significantly facilitated with a source code analyser tool. Such a tool is even more powerful, if it provides a full-fledged query language to express queries over the static analysis information collected by the tool. This paper recommends an approach, and good practices, for the design of a query language for static source code analysis, with aims to intuition, consistency and reliable implementation. The approach is presented for querying information about Erlang source code. The characteristics of the proposed query language are its purely functional style, easy-to-learn syntax, and its formal definition in Haskell.

1 Introduction

Software developers use various programming language processing tools, which are invaluable during the development and maintenance of reliable software and high-quality source code. Such tools have different capabilities, ranging from style checking (e.g. coding conventions) to bug forecast to complexity measures to bad smell detection to code comprehension. Code comprehension tools allow users discover and investigate dependencies among source code entities, and help understand how the small pieces fit together.

Typically, the behaviour of code comprehension tools boils down to two phases. First, an *analysis phase* builds an internal data structure (e.g. a graph). Users then collect information from the data structure (e.g. traverse the graph) in an *exploration phase*. When it comes to the exploration phase, the flexibility of the user interface is greatly enhanced with an expressive query language. An easy-to-learn and intuitive query language can significantly reduce the time needed to gather information about a program. For instance, using the query language, a programmer is able to trace back the origin of an invalid actual parameter in the source code, or search for leftover debugging messages before software release. In the end, programmers can understand the source code faster with the query language, because they are not restricted to a set of predefined queries; instead, they can easily write queries that yield exactly the desired information.

This paper reports on our experiences with the design of a query language for static source code analysis. This query language is specific to a programming language: information about source code written in that particular language can be collected with the query language. On the one hand, we believe that the usability of the query

Copyright © by the paper's authors. Copying permitted for private and academic purposes.

In: E. Vatai (ed.): Proceedings of the 11th Joint Conference on Mathematics and Computer Science, Eger, Hungary, 20th – 22nd of May, 2016, published at http://ceur-ws.org

language is improved due to its dependence on the programming language under consideration. On the other hand, similar programming languages yield similar query languages, so the technique presented here is quite generic and applicable to a wide range of programming languages.

The programming language used for our query language is Erlang [Arm13]. A brief summary of Erlang is given in Section 2.1. However, the code examples presented in the paper are quite straightforward, and can be understood with general programming skills, without prior knowledge of Erlang. Our query language is, in fact, a domain-specific language for traversing graphs with labelled nodes and edges, which represent static (i.e. compile-time) information about some Erlang source code.

The main contributions of this paper are a query language design practice and a systematically designed query language that has the following properties:

- The language is simple and intuitive. Queries make use of the set builder notation, which is well-known for everyone from high-school mathematics. Most of the syntax (function application and list comprehension, the latter also known as ZF-expression, after Zermelo-Frankel set theory) is borrowed from the Haskell programming language [Jon03], and is fairly lightweight. This idea of relying on Haskell syntax is proven to be helpful in creating a simple, easy-to-extend language. The syntax with descriptive examples is presented thoroughly in Section 3.

- The language is strongly typed, which makes it possible to report errors in queries before execution. Furthermore, programmers get friendly and self-explanatory error messages. Typing is discussed in more detail in Section 4.

- The language has well-defined semantics. A denotational semantics has been designed for the language, and expressed in Haskell, the implementation language of our choice. In Haskell, the executable denotation semantics of language features can easily and clearly be defined. The formal semantics of the query language, however, is out of the scope of this paper.

- The implementation of the language is reasonably fast in spite of the fact that it has to communicate with the back-end of a code comprehension tool written in Erlang, and thus it has to take care of the (de-)serialization of Erlang data. Section 6 reports on our related experiences.

- The language is lazy, which facilitates minimizing communication between the Haskell front-end and the Erlang back-end. Thanks to `pipes`[1], a stream processing library for Haskell, practically any function could easily be turned into lazy.

2 Preliminaries

Before presenting our query language for collecting static source code analysis information on programs written in Erlang, some introduction is given to the necessary background. First the main characteristics of the Erlang programming language are highlighted (Section 2.1). Then RefactorErl [BHH+11], a source code analysis and transformation tool for the Erlang language is briefly described (Section 2.2). RefactorErl serves as the back-end to our query language, providing the static source code analysis information to work on. Finally, Section 2.3 explains how querying in RefactorErl looked like earlier, and why it was necessary to design the query language described in this paper.

2.1 What is Erlang?

Erlang is a general purpose dynamically typed functional programming language with built-in primitives for concurrency, distribution and message passing communication. The language was originally designed for the telecommunication application domain. Nowadays, the Erlang Open Telecom Platform (OTP) framework with the Erlang Virtual Machine (VM) is used for building highly available, robust, fault tolerant systems in various domains. The Erlang source files are compiled to bytecode (BEAM), which run on the Erlang VM. The VM is responsible for process scheduling and garbage collection as well.

There are several built-in data types in Erlang. Besides the primitive data types (e.g. number, atom etc.) there are four compound data types, namely tuple, list, record and map. The usual constructs for functional

[1] `https://hackage.haskell.org/package/pipes`

programming, i.e. pattern matching, branching expressions, recursion (with tail-call optimization), higher-order functions and lambda-expressions are also available in this call-by-value, impure functional language.

Erlang is a single assignment language, that is, once a variable is bound it cannot be changed. Variables can be bound with match expressions (e.g. `Palindromes = ...`) and with pattern matching. Note, however, that if a bound variable occurs either in a match expression or a pattern, the construct behaves as an equality assertion. Variable names always start with a capital letter, while function names and so called atoms (for instance boolean values `true` and `false`) always start with a lowercase letter.

The most important building block of Erlang is the *function definition*. Similarly to many other functional languages, functions can be defined with pattern matching, and hence the definition of a function may be split up into multiple *function clauses*. The body of a function is a sequence of expressions, the last of which providing the result of the function. Function names can be overloaded on the arity, therefore we refer to functions with `name/arity` syntax, such as `is_palindrome/1`.

Function definitions are organized into *modules*. A module can export functions: these functions can be called from other modules. A module can also import functions from other modules, which allows those functions to be called like functions defined in the same module. Otherwise, a function from another module must be called with a module qualifier, such as in `lists:reverse([1,2,3])` (reversing the list `[1,2,3]` by calling the `reverse/1` function from the standard library module `lists`). Erlang OTP also supports the organization of modules into applications, which are reusable components that can be started and stopped as single units. An Erlang OTP system may contain multiple applications.

As an example, let us consider the following Erlang code. The `palindrome:main/0` function prints out `racecar` and `madam`, the two palindromes from the three words tested.

```
-module(palindrome).
-export([main/0]).

main() ->
  Palindromes = [Word || Word <- ["racecar", "owl", "madam"], is_palindrome(Word)],
  io:format("Palindromes: ~p", [Palindromes]).

is_palindrome(Text) ->
  Text == lists:reverse(Text).
```

The Erlang compiler has a preprocessor, which allows us to define and use macros, as well as to textually include header files, as we shall see in forthcoming examples.

2.2 What is RefactorErl?

RefactorErl is a static source code analysis and transformation tool for Erlang with the following main features:

- a semantic query language, to gather information about the source code
- more than twenty (semantics preserving) refactoring transformations
- computation and visualisation of dependencies
- duplicated code detection
- investigation tracking
- component clustering

The functionality of RefactorErl is available through several interfaces. It has a web-based graphical user interface, plug-ins for several editors, and a command-line interface as well.

RefactorErl represents the source code of an Erlang program in a graph data structure with labels attached to nodes and edges. This data structure, the *Semantic Program Graph* (SPG) contains the abstract syntax tree, as well as *semantic information* (like variable scoping, function call graph, data-flow, process structure etc.) and *lexical information* (sufficient for reproducing the source code verbatim). Hence, the SPG is made up of lexical, syntactical and semantical nodes, and lexical, syntactical and semantical edges.

The SPG is an extremely rich representation of source code. The Erlang module shown in Section 2.1 generates a Semantic Program Graph that would not fit on one page. Figure 1 shows a fragment of the SPG, focusing on the semantic information related to the `is_palindrome/1` function only. Semantic nodes and edges are

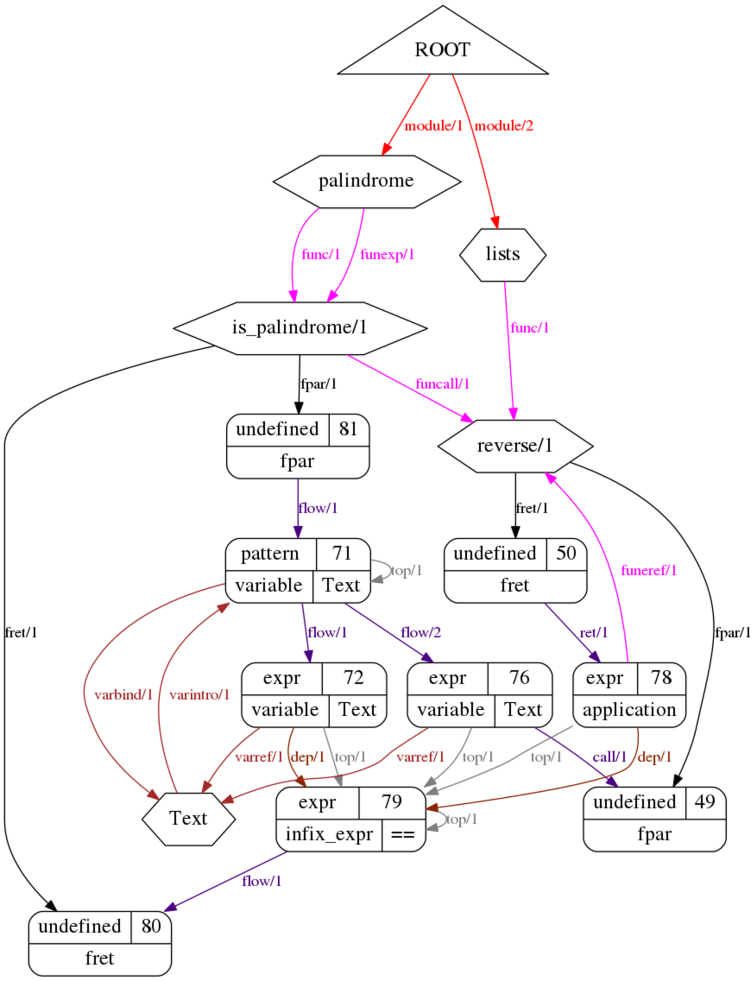

Figure 1: Semantic and syntactic nodes and edges of the SPG of `is_palidrome/1`.

drawn as hexagons and coloured edges, respectively. Some syntactic nodes and edges are also included, they are represented by rounded rectangles and black edges. Lexical information is completely omitted here.

The collection of information on the Erlang source code can be accomplished by traversing the SPG. Such a traversal may start from the `ROOT` node, and visit other nodes through the labelled edges relevant from the point of view of the searched information. Working directly with the data structure, however, may be uncomfortable for an Erlang software developer. Therefore, the *Semantic Query Language* was designed [TBKH] to provide a user-level querying interface for RefactorErl. The queries refer to Erlang language concepts, thus it is easy to use for Erlang developers even without any knowledge on the internal structure of SPGs. Unfortunately, as Section 2.3 reveals, the Semantic Query Language has a number of drawbacks, therefore the design of a new query language, *NequeLace*, became necessary.

2.3 The Need for a New Query Language

First of all, let us have a glance at the original Semantic Query Language. The basic building blocks in this language are the *selectors*, the *filters* and the *properties*. Selectors identify semantic entities, such as modules, functions, variables etc., and they can be combined with the *dot* operator to express containment with respect to program structure. The result of the selection is a set of semantic nodes in the SPG. For example, to select all the functions from all of the modules, the query `mods.funs` can be used. Filters narrow the result set based on some predefined properties. For example, we can filter out expressions based on their type and possible values. Filters are denoted with square brackets. To obtain all the `foo` functions from all of the modules, we can write the following query: `mods.funs[name=foo]`. Other advanced features, like *iterations*, *closures*, *property queries*, *variables*, *set operations* etc. allow us to write more advanced queries. For instance, we can check whether a

function `foo` depends on another function `bar` with the following query ("+" denotes the transitive closure of a selector, in this case that of `calls`, which is the selector to operate on the function call graph).

```
mods.funs[name=foo and .(calls)+[name=bar]]
```

The Semantic Query Language has turned out to be suboptimal in two major aspects: maintainability of the implementation and readability. Let us consider first the maintainability of the implementation. Although the language was designed to be easily extensible with new selectors and properties, adding new language features is not at all straightforward. For example, originally the query language did not have variables to store temporarily the results of sub-queries. When the decision has been made to add them to the language, it turned out that variables cannot be implemented orthogonally, they affect all other features of the language. Finally, it took cca. 5 person months to implement this feature. This was considered as a clear sign of the necessity of a complete language redesign.

The other problematic aspect of the Semantic Query Language is readability. The queries expressed in this language tend to be very terse, which can have a negative impact for average users. Furthermore, sometimes quite strange, or ambiguous, expressions arise, for instance when property names overlap with atoms used in the analysed Erlang programs. Consider the following record declaration in an Erlang program, which introduces a new record type, the name of which is `name` (and the field names are `surname` and `given_name`).

```
-module(person).
-record(name, {surname, given_name}).
```

A query which collects every reference to this record would look as follows.

```
mods[name=person].records[name=name].refs
```

The filter part of the query, `name=name`, can easily be (mistakenly) seen as a tautology. The advanced user of the Semantic Query Language, of course, will know that the first occurrence of `name` is the name of a property (an edge label in the SPG), while the second occurrence is a value used in the analysed Erlang program. To avoid confusion for average users, however, a syntax that makes clear distinction between the two uses of the same word would be advantageous. For example, a list comprehension expression in Erlang would use the following syntax to describe this query.

```
[Refs || M <- modules(), name(M) == "person",
         R <- records(M), name(R) == "name",
         Refs <- references(R)]
```

This more verbose syntax is probably more readable for most software developers. There is no *dot*-operator (which can also be mistakenly interpreted as a record field selection for novice users), and the ambiguities illustrated above are also resolved. This observation influenced the design of the syntax for NequeLace, the new query language.

3 NequeLace: the New Query Language

The set builder notation is often used in mathematics classes in high-school, therefore it excels at being the fundament of the syntax of a query language. List comprehensions, providing similar notation, are widely used in functional programming languages (including Erlang), although with slightly different concrete (or surface) syntax in different languages. Set builders in NequeLace resemble most to Haskell list comprehensions.

The next important language construct in NequeLace is function application. Again, we opted for the Haskell-like, juxtaposition syntax for this construct. The core of the language is defined as a set of functions, many of them corresponding to the selectors of the Semantic Query Language. These functions are all provided by RefactorErl, which acts as the back-end for the query languages.

We shall discover NequeLace through a series of interesting example queries. Let us consider again the palindrome module.

```
-module(palindrome).
-export([main/0]).

main() ->
  Palindromes = [Word || Word <- ["racecar", "owl", "madam"], is_palindrome(Word)],
  io:format("Palindromes: ~p", [Palindromes]).
```

```
is_palindrome(Text) ->
  Text == lists:reverse(Text).
```

The `lists:reverse/1` function operates on lists. If the argument is of some other type, a runtime exception is raised. Since Erlang is dynamically typed, such exceptions may show up quite often. This happens, for instance, if we replace the string literal `owl` with the number literal `3.1415927`. In order to find and correct a type-related bug, the software developer may want to know all the possible values which are used as actual parameters to a given function. The following query traces back origins of the parameter `Text` of the `is_palindrome/1` function above.

```
{(p,o) | m <- modules, name m == "palindrome",
         f <- functions m, name f == "is_palindrome",
         p <- parameters f,
         o <- origin p}
```

The result of the query is a set of pairs, where the first element of the pair is (the SPG node representing) the parameter `Text`, and the second element is (the SPG node representing) an Erlang expression that can flow into this parameter. If `owl` is replaced with `3.141592`, the result of the query is the following.

```
  p     o

Text  Word
Text  "noon"
Text  3.141592
Text  "racecar"
Text  Word
Text  Text
```

In terms of the semantic program graph in Figure 1, a query performs a depth-first traversal, starting from the `ROOT` node. The `modules` function selects and traverses outgoing edges labelled `module`. For each module, the `functions` function selects and traverses edges labelled `func`.

Some functions exhibit a more complex behaviour. The `origin` function takes the node of an Erlang expression or parameter (e.g. parameter `Text`), and traverses edges labelled `flow` in opposite direction transitively. This way, it reaches points in the code where the value of the expression or parameter originates from.

Before introducing the next example, we extend the analysed Erlang codebase with the function `prop_palindrome/0`, which tests the correctness of `is_palindrome/0`. It uses `PropEr` [PS11], a testing tool, to randomly generate test input to `is_palindrome/0`.

```
-include_lib("proper/include/proper.hrl").   % textually include a header file which
                                             % provides definition to the FORALL macro etc.
prop_palindrome() ->
  ?FORALL(Chars, list(range($a, $z)),
          is_palindrome(Chars ++ lists:reverse(Chars))).
```

Let us imagine for a moment that a `PropEr` upgrade introduces API changes (for instance, `range/2` now excludes the upper bound, and `list/1` is renamed to `list_of/1`). Applications of `PropEr` functions which are involved in API changes need adjustments. For this reason, a systematic approach which finds all applications of functions of some module becomes necessary. Lexical search (e.g. using `grep`) which looks for a module name in the source code does not work properly, because imported functions may be invoked without qualification, similarly to `list/1` above (the `include_lib` directive automatically imports many commonly used functions, including `list/1`). In this case, semantic analyses offer a better alternative. Once the code is analysed, the query that collects all invocations of `PropEr` functions may be formulated as follows. We can filter functions by name if we are looking for a specific function (such as `list/1`).

```
{(f, app) | m <- modules,
            m ~= "^proper",
            f <- functions m,
            app <- references f}
```

The operator ~= matches a regular expression against a string. In this example, it is used to filter out modules which are not part of the `PropEr` library, i.e. names of which do not start with `proper` (note that in our regular expression the symbol ^ matches the beginning of a string).

As the third example, consider again the problem of collecting references to the record called `name` from Section 2. The NequeLace query for this problem can be written in the following way.

```
{refs | m <- modules, name m == "person",
        r <- records m, name r == "name",
        refs <- references r}
```

Set builders are not the only form of queries. Function applications per se are also valid. As an extreme example, consider a query to list every module in the analysed code.

```
{m | m <- modules}
```

A query with the same effect can be written without the set-builder notation as well.

```
modules
```

More interesting examples of queries without set builders will appear in Section 3.1. In NequeLace, there is a definite improvement on the effort needed to understand queries compared to the Semantic Query Language. However, there is another area where the new query language falls a little short: compactness. Writing a query which spans many lines tends to be tedious. Fortunately, user-defined functions can address this problem, as discussed in Section 3.2. The full syntax of NequeLace is presented in Section 3.3 in more detail.

A subtle point in the above queries is that functions such as `name` and `references` may be applied to values of various types. This is because these are polymorphic functions. Section 4 elaborates on the type system.

3.1 Implicit query parameters

It must be clear from the examples presented so far that queries are implicitly parametrized with an SPG, namely the semantic program graph of the Erlang code which is loaded into an instance of RefactorErl. A NequeLace interpreter connects to, and communicates with, a running instance of RefactorErl, and the queries are executed against the SPG loaded into that particular instance. Moreover, NequeLace is typically used inside a program development environment (such as an IDE or an editor), which can provide a context to queries. Most notably, queries can make use of code position information. To achieve this, the implementation of NequeLace should be able to cooperate with the given program development environment, e.g. through an IDE plug-in. If a certain row/column position of a certain source file is selected in the editor, the position information can be passed to the NequeLace interpreter, and can be used to identify an SPG node in the back-end.

With the above mechanism, NequeLace supports *position-based queries*, viz. queries that rely on implicit position information. We can refer, for example, to the Erlang function opened in our editor when the query is executed using the `atFunction` primitive of NequeLace (or `@fun` in the original Semantic Query Language). To refer to the record selected in our editor, the `atRecord` primitive can be used. If the current position in the editor is not a record, an error message is generated by `atRecord`. One can utilise this feature to find all references to a given record (the one selected in the editor) with the following code, which also gives a nice example of a query not using set-builder notation.

```
references atRecord
```

This query is not only shorter than the one using set-builder notation, but also slightly more effective. On the one hand, the query with the set-builder notation starts from the `ROOT` node, and proceeds towards the record declaration node along edges in the semantic program graph. Once found, it collects expression nodes at the end of incoming `recref` edges. The position-based query, on the other hand, starts directly from the record declaration node, and can avoid searching for the record in the SPG.

3.2 Query Functions

In order to avoid long queries, and to encourage query reuse, user-defined functions have been introduced into NequeLace. There are at least two use cases where such *query functions* excel. First, the programmer may create a function from many similar, frequently used queries, and so the function acts as a parametric query. Second, a function may be defined as a shorthand for a standard library function.

A query function is very similar to a function in Haskell, and in the same time it is considerably simpler, since it lacks several language features. Namely, pattern matching, branching, local bindings in a `let` or `where` clause and recursion have so far turned out to be unnecessary, and hence are currently not supported.

As an example, consider function `lookup`, which returns a set of functions with a specified name and module. There could be more than one such function, as overloading is allowed in Erlang. Function `params` returns the set of parameters of a function specified by module name, function name and number of parameters.

```
lookup mod fun    = {f | m <- modules, name m == mod,
                         f <- functions m, name f == fun}

params mod fun n = {p | f <- lookup mod fun, arity f == n,
                         p <- parameters f}
```

Similarly to many declarative languages, such as Erlang and Haskell, there is no need to specify the types of parameters. The type inference algorithm, in these cases, can infer that the type of both `mod` and `fun` is string, and `n` is an integer.

User-defined functions can be saved into NequeLace source files. The `with` keyword followed by a filename brings the function definitions stored in the file into scope. For instance, we can save the `lookup` and `params` functions in the file `functions.sq`, and then make the first example from the previous section shorter.

```
with "functions.sq"
  {o | p <- params "palindrome" "is_palindrome" 1
       o <- origin p}
```

3.3 Formal Syntax

The core of NequeLace is very small, and contains only essential features: iteration over sets, and function application. All other language features are implemented as functions, laid on top of the core. This makes the query language easy to extend: when a new language feature is needed, it is presumably sufficient to define one or more functions — it is unnecessary to modify the parser (unless we introduce an infix operator), change the type inference algorithm or the evaluation.

Following the principles advocated by APL [Ive62] and other, more modern languages such as Agda [BDN09], Unicode symbols are also included in the syntax. Consequently, queries become terse and intuitive at the same time, especially in the case of set operations. The operators $\cup$, $\cap$, $\in$, $\subseteq$, and the function composition operator $\circ$ are all valid infix binary operators. With a modern text editor it should not impose any problems to enter these symbols, and we can also introduce ASCII synonyms (such as `union` for $\cup$) using user-defined functions.

The abstract syntax of NequeLace is as follows.

e	:=	n	numeric literal
	|	*str*	string literal
	|	dataConstr *str*	data constructor
	|	$(\vec{e})$	tuple
	|	v	identifier
	|	$e\ e'$	function application
	|	$\lambda v.e$	function abstraction
	|	e >>= e'	bind
	|	return e	lift
	|	guard e	filter
	|	with $\vec{f}\ e$	function import
f	:=	$v = e$	function definition

Data constructors are represented as plain strings, and the type of a data constructor is determined during type checking (see Section 4 for details). Furthermore, operations are represented as ordinary function applications. Also, the nodes "bind", "lift" and "filter" constitute the backbone of set builder expressions. The way set builders are parsed into combination of these nodes is similar to the way Haskell list comprehensions could be written using >>=, `return` and `guard`. Let us consider the query function from Section 3.2:

```
query        = with | expression;
with         = 'with', string literal, expression;
expression   = application
             | operation       | tuple
             | identifier      | constructor
             | numeric literal | string literal
             | set builder     | '(', expression, ')';
application  = expression, expression;
operation    = expression, operator, expression;
operator     = '==' | '/=' | '<' | '=<' | '>' | '>='
             | '~=' | '∪'  | '∩' | '∈'  | '⊆' | '+'
             | '-'  | '*'  | '∘' ;
identifier   = lowercase letter, {letter | number | '_'};
constructor  = uppercase letter, {letter | number | '_'};
set builder  = '{', expression, '|', builder elem, {',' builder elem}, '}';
builder elem = generator | expression;
generator    = identifier, '<-', expression;
function     = identifier {identifier} '=' expression
```

Figure 2: EBNF specification of the concrete syntax

```
lookup mod fun == {f | m <- modules, name m == mod,
                       f <- functions m, name f == fun}
```

The abstract syntax of the function body is the following. We regard >>= as a right-associative binary operator.

```
modules >>= λm.
guard ((==) (name m) mod) >>= λx.
functions m >>= λf.
guard ((==) (name f) fun) >>= λy.
return f
```

Note that the parameters `x` and `y` are introduced because patterns, such as wildcard pattern, are not supported by the syntax. The type of both parameters is the unit type.

The formal EBNF specification of the concrete syntax is shown in Figure 2. It is worth noting that the syntax is fairly general. It seems as if it is taken from a general-purpose high-level functional language like Haskell. Apart from the operators it does not contain anything related to RefactorErl or Erlang, which makes it highly reusable. When another query language in different context is needed, all needed to do is replacing the set of built-in functions. Therefore, even the parser can be reused with slight modifications, e.g. replacing the set of infix operators.

4 Type Checking Queries

The language is designed to be strongly typed from the ground up. Types are inferred, so it is unnecessary to specify types of expressions and functions, which leads to a more convenient and compact query language. Type inference is performed before evaluation, and an error is reported if type inference does not succeed. A separate typing mechanism is not only necessary in case of a stand-alone language such as NequeLace, but also useful in presenting friendly error messages to the user. We give examples of error messages at the end of this section.

This section elaborates on the type inference mechanism that supports bounded and unbounded parametric polymorphism. The type system is based on System F [Har16], and it additionally includes type classes. We rely Simon Peyton Jones and other's work [HHJW94] on formal definition of Haskell type classes.

Figure 3 shows the syntax of types. There are three variants of types: simple, polymorphic and bounded polymorphic. In a polymorphic type the value of a type variable may or may not be constrained, depending on the context. Quantified type variables in the context have a constrained set of values, whereas the rest of quantified type variables have no such constraints. For instance, the function `null`, which checks whether a set has any element, is polymorphic. Its type is written as follows in Haskell notation:

```
null :: Set a -> Bool
```

Type variable	α		
Type constructor	χ		
Type class	κ		
Context	θ	$:=$	$\langle \kappa\ \alpha \rangle$
Type	σ	$:=$	α
		$\mid$	$\chi\ \sigma_1 \ldots \sigma_n$
		$\mid$	$\sigma' \to \sigma''$
		$\mid$	$\langle \kappa\ \sigma \rangle \Rightarrow \sigma'$
Polymorphic type		$\mid$	$\forall \alpha . \theta \Rightarrow \sigma$

Figure 3: Syntax of Types

It means that `null` takes a set as argument, and the type of elements in the set (the type variable `a`) may be chosen freely. In contrast, the type of `name` is the following:

```
name :: Named a => a -> String
```

In this polymorphic type `Named` is a type class, `a` is a type variable, and `String` is a type. The type variable `a` is constrained to the types that belong to the `Named` class (e.g. `Module` and `Function`). This is indicated by the `Named a` context.

In this section we use a slightly different notation in which Greek letters denote all types except type constructors, to be consistent with Figure 3. We also omit the pair of angle brackets when the context is empty. For instance, types of `null` and `name` are written as follows:

$$\texttt{null} : \forall \alpha .\ \textbf{Set}\ \alpha \to \textbf{Bool}$$
$$\texttt{name} : \forall \alpha .\ \langle \textbf{Named}\ \alpha \rangle \Rightarrow \alpha \to \textbf{String}$$

In order to make type inference feasible, several environments are introduced. The following table summarizes the environments, and gives examples for each environment. During type inference these environments are populated with more elements, of course.

Environment	Notation	Example
Type variables	$TVar$	$\{\alpha,\ \beta : \textbf{Named}\}$
Free variables	Var	$\{x : \alpha, y : \textbf{Module}\}$
Type constructors	$TCons$	$\{\textbf{String} : 0,\ \textbf{Int} : 0,\ \textbf{Set} : 1\}$
Type classes	$TClass$	$\{\textbf{Named},\ \textbf{Ord}\}$
Functions	Fun	$\left\{\begin{array}{ll} \texttt{functions} : & \textbf{Module} \to \textbf{Set Function} \\ \texttt{null} : & \forall \alpha .\ \textbf{Set}\ \alpha \to \textbf{Bool}, \\ \texttt{name} : & \forall \alpha .\ \langle \textbf{Named}\ \alpha \rangle \Rightarrow \alpha \to \textbf{String} \end{array}\right\}$

Roles of the environments may be summarized as follows.

- The type variable environment stores free type variables. If a type variable has a constrained set of values, then the environment associates the type variable with a type class.
- The free variable environment associates each free variable with a type.
- The type constructor environment maps type constructors (such as **String** and **Set**) to the number of their type arguments. It has a fixed size, since user-defined types are not supported.
- The type class environment comprises the pre-defined type classes. It also has a fixed size, since user-defined classes are not supported.
- The function environment associates all known (built-in and user-defined) functions with their type. Initially, it consists of the standard library, and user-defined functions are added when the `with` construct is used.

In the typing rules $E\ name = inf$ denotes that the environment E associates $name$ to the information inf. If the information is not relevant, and we are only interested in whether the $name$ is in the environment, we write $E\ name$.

$$\frac{(\Gamma.TVar)\,\alpha}{\Gamma \vdash \alpha}\ [\textsc{Type}\ \alpha] \qquad \frac{(\Gamma.TCons)\,\chi = k \quad \Gamma \vdash \sigma_i}{\Gamma \vdash \chi\ \sigma_1 \ldots \sigma_n}\ [\textsc{Type}\ \chi]$$

$$\frac{\Gamma \vdash \sigma \qquad \Gamma \vdash \sigma'}{\Gamma \vdash \sigma \to \sigma'}\ [\textsc{Type Arrow}] \qquad \frac{(\Gamma.TClass)\,\kappa_i \quad \Gamma \vdash \sigma \qquad \Gamma \vdash \sigma'}{\Gamma \vdash \langle \kappa\ \sigma \rangle \Rightarrow \sigma'}\ [\textsc{Type Bounded Poly}]$$

$$\frac{(\Gamma.TClass)\,\kappa \quad TVar = \{\alpha : \kappa\} \quad \Gamma \oplus TVar \vdash \sigma}{\Gamma \vdash \forall \alpha.\, \langle \kappa\ \alpha \rangle \Rightarrow \sigma}\ [\textsc{Type Context}] \qquad \frac{TVar = \{\alpha\} \quad \Gamma \oplus TVar \vdash \sigma}{\Gamma \vdash \forall \alpha . \sigma}\ [\textsc{Type Poly}]$$

Figure 4: Rules for Types

$E_1 \oplus E_2$ denotes the combination of the environments E_1 and E_2. It also checks that domains of the environments are disjoint, so the function is well defined.

$$(E_1 \oplus E_2)\, var = \begin{cases} E_1\ var & \text{if } var \in dom\,(E_1) \text{ and } var \notin dom\,(E_2) \\ E_2\ var & \text{if } var \in dom\,(E_2) \text{ and } var \notin dom\,(E_1)\,. \end{cases}$$

Let us introduce a compound environment Γ as tuple of the aforementioned environments, which let us avoid writing all environments in inference rules. The dot operator selects one of the components, for example $\Gamma.TVar$. We also use the operator $\oplus$ to add new elements to a component of Γ, in order to save space. For example, considering $V = \{x : \sigma'\}$, $\Gamma \oplus Var$ denotes a new compound environment Γ', and the environment $\Gamma'.Var$ now includes a new variable x.

The rules for types in Figure 4 ensure that types are syntactically valid. The rule Type α requires that free type variables are found in the environment, and the rule Type κ requires that type constructors are fully applied. Rules Type Context and Type Poly together describe how polymorphic types are made. In Type Context, $(\Gamma.TClass)\,\kappa$ means the type class κ is known, and $TVar = \{\alpha : \kappa\}$ means the type variable α is constrained to instances of the type class κ. Type Poly specifies types with empty context. Both rules removes quantified type variables from the environment.

Syntax-directed rules for expressions and function definitions are shown in Figure 5. There are two Constr rules for data constructors, because there are two enumeration types built into NequeLace. One is **ExprType**, which has data constructors such as `Variable` and `Application`. The other is **Recursivity**, which represents function categories, and has data constructors `NonRecursive`, `NonTailRecursive` and `TailRecursive`. The set of data constructors are disjoint, so a simple parsing is able to tell the type of a data constructor. A query with data constructor is presented in Section 5.2. The rules Bind, Return and Guard specializes the types of `>>=`, `return` and `guard` in the Haskell standard library [Jon03] to the type constructor **Set**. The rule Import encodes the scoping rules of function definitions. Each function may use already defined functions, which rules out recursion. This technical restriction makes the implementation straightforward, and it is not a real trade-off, since recursive functions in queries are hardly needed. The rule Context introduces qualified expressions, and the rule Inst eliminates them. In Inst, the premise

$$\Gamma \overset{\text{inst}}{\vdash} \kappa\ \sigma$$

means that the type σ is an instance of the type class κ. The rules Poly Abs introduces quantification, creating a polymorphic expression with an empty context, and the rule Poly Appl eliminates quantification by substitution. Note that by substituting a type for a type variable in Poly Appl, we get a bounded polymorphic type, which will be eliminated by the rule Inst.

There are definite advantages of creating a type inference mechanism. As with most statically typed languages, type errors are revealed in the early stages of processing. Furthermore, by using own type inference method, programmers get more friendly error messages. For example, in the following query the actual parameter of `functions` is intentionally omitted, thus the type inference reports an error:

```
{ f | m <- modules, f <- functions }
```

$$\frac{}{\Gamma \vdash n : \mathbf{Int}}\ [\textsc{Num}] \qquad \frac{}{\Gamma \vdash str : \mathbf{String}}\ [\textsc{String}]$$

$$\frac{}{\Gamma \vdash \text{dataConstr } str : \mathbf{ExprType}}\ [\textsc{Constr}] \qquad \frac{}{\Gamma \vdash \text{dataConstr } str : \mathbf{Recursivity}}\ [\textsc{Constr}]$$

$$\frac{\Gamma \vdash e_i : \sigma_i}{\Gamma \vdash (e_1, e_2, \ldots, e_n) : (\sigma_1, \sigma_2, \ldots, \sigma_n)}\ [\textsc{Tuple}]$$

$$\frac{(\Gamma.Var)\, v = \sigma}{\Gamma \vdash v : \sigma}\ [\textsc{Ident}] \qquad \frac{(\Gamma.Fun)\, v = \sigma}{\Gamma \vdash v : \sigma}\ [\textsc{Fun}]$$

$$\frac{\Gamma \vdash E : \sigma \to \sigma' \qquad \Gamma \vdash F : \sigma}{\Gamma \vdash EF : \sigma'}\ [\textsc{Appl}] \qquad \frac{\begin{array}{c} Var = \{x : \sigma\} \\ \Gamma \oplus Var \vdash E : \sigma' \end{array}}{\Gamma \vdash \lambda x.E : \sigma \to \sigma'}\ [\textsc{Abs}]$$

$$\frac{\begin{array}{c} \Gamma \vdash e : \mathbf{Set}\ \sigma \\ \Gamma \vdash e' : \sigma \to \mathbf{Set}\ \sigma' \end{array}}{\Gamma \vdash e\ \texttt{>>=}\ e' : \mathbf{Set}\ \sigma'}\ [\textsc{Bind}] \qquad \frac{\Gamma \vdash e : \sigma}{\Gamma \vdash \text{return } e : \mathbf{Set}\ \sigma}\ [\textsc{Return}]$$

$$\frac{\Gamma \vdash e : \mathbf{Bool}}{\Gamma \vdash \text{guard } e : \mathbf{Set}\ ()}\ [\textsc{Guard}] \qquad \frac{\begin{array}{c} \vec{f} = (f_1, \ldots, f_n) = ((v = e)_1, \ldots, (v = e)_n) \\ Fun_i = \{v_j : \sigma_j\},\ j < i \\ \Gamma \oplus Fun_i \vdash f_i : \sigma_i \qquad \Gamma \oplus Fun_{n+1} \vdash e : \sigma \end{array}}{\Gamma \vdash \text{with } \vec{f}\ e : \sigma}\ [\textsc{Import}]$$

$$\frac{\begin{array}{c} (\Gamma.TClass)\, \kappa_i \\ TVar = \{\alpha : \kappa\} \\ \Gamma \vdash \kappa\ \alpha \qquad \Gamma \vdash E : \sigma \end{array}}{\Gamma \vdash E : \forall \alpha.\, \langle \kappa\ \alpha \rangle \Rightarrow \sigma}\ [\textsc{Context}] \qquad \frac{\Gamma \vdash E : \langle \kappa\ \sigma \rangle \Rightarrow \sigma \qquad \Gamma \overset{\text{inst}}{\vdash} \kappa\ \sigma}{\Gamma \vdash E : \sigma}\ [\textsc{Inst}]$$

$$\frac{\begin{array}{c} TVar = \{\alpha\} \\ \Gamma \oplus TVar \vdash E : \sigma \end{array}}{\Gamma \vdash E : \forall \alpha.\sigma}\ [\textsc{Poly Abs}] \qquad \frac{\Gamma \vdash E : \forall \alpha.\theta \Rightarrow \sigma \qquad \Gamma \vdash \sigma}{\Gamma \vdash E : (\theta \Rightarrow \sigma)\,[\alpha := \sigma]}\ [\textsc{Poly Appl}]$$

$$\frac{\Gamma \vdash e : \sigma}{\Gamma \vdash v = e : \sigma}\ [\textsc{FunDef}]$$

Figure 5: Typing Rules of Expressions and Function Definitions

When the user tries to interpret the query, she gets an error message. Here {a1} denotes a set of arbitrary elements, and {Function} denotes a set of functions.

```
type error: expected: {a1},
            actual:   Module -> {Function}
in expression "functions"
```

Moreover, the user an gets easy-to-understand error message when the formal argument type differs from the actual argument type. For instance, the expression `not 2` does not have type as the error message indicates:

```
type error: expected: Bool,
            actual:   Int
at the 1. argument of "not"
```

In summary, using the type inference method presented in this section, we are able to report good error messages for the programmers, thus making the language more friendly.

5 Using Semantic Queries

We present examples of semantic queries that solve real-life problems. The following experiments are done with the source code of *mnesia* (version 20.1), an open source and built-in data base management system of the Erlang OTP.

To understand the behaviour of a certain function, we need to understand the behaviour of the called functions as well. Thus, a query that helps us to locate the functions called by the given function is quite useful. We can express this in both query languages for a given function `mnesia:stored_key/6`.

```
{c | m <- modules, name m == "mnesia",
     f <- functions m, name f == "stored_keys",
     c <- calls f}
```

The same query in Semantic Query Language:

```
mods[name=mnesia].funs[name=stored_keys].calls
```

The result set contains the following eight functions.

```
mnesia:abort/1
mnesia:next/2
mnesia:prev/2
mnesia:ts_keys/5
mnesia:get_ordered_tskey/3
mnesia:get_next_tskey/3
ets:match/2
lists:last/1
```

To prove the usability of NequeLace, we have studied a few queries that was suggested by problems of our industrial partners.

5.1 Dynamic calls

Let us consider the previous query, where we selected the functions called by `mnesia:stored_key/6`. In the body of this function we can find dynamic function calls as well. For example, in line 994 in the module `mnesia`, we find the expression `mnesia:Op(Tab,Key)`.

This makes the understanding of the function even harder since we do not know what functions are referenced by the variable `Op`. Therefore our first use case scenario tries and answers the following question: what are the functions referenced dynamically at a certain point in the program?

This query can be expressed by NequeLace. The following position-based query relies on implicit position parameter.

```
dynamicFunctions (topExpr atExpr)
```

The result shows us that the dynamic application may refer to the functions `next` or `prev`:

```
mnesia:next/2
mnesia:prev/2
```

We provide the same query in the Semantic Query Language, which is used to benchmark the performance in Section 6.1. In a text editor, or in the web interface of RefactorErl, we can select the aforementioned expression and run the following query:

```
@expr.top.dynamic_function
```

When no graphical interface is available, a simple command line query can select the expression by a regular expression:

```
ri:q(mnesia, "mnesia:Op\\(Tab,Key\\)", "@expr.top.dynamic_function").
```

A third option to obtain the desired information is to select all dynamic references of a function. However, if there are multiple dynamic calls the result set is bigger:

```
mods[name=mnesia]
  .funs[name=stored_keys]
  .expressions.sub[type=application]
  .dynamic_functions
```

5.2 Record Field Updates

Another main concern of Erlang developers is the identification of records. When we are talking about records, the name of the record is not unique in the program. Distinct files may declare distinct records with the same name. Therefore, when we are performing a text search on a record name, we will find plenty of false positive hints.

Another problem is the identification of certain record fields. The name of the record and the reference of the field may be placed in consecutive lines in the source code. This makes the identification of a record field even harder with line based regular expression search as well. However, RefactorErl is not sensitive to location (i.e. line) information. The tool identifies entities (e.g. records) based on semantic analysis.

The usage of records is quite common when we are implementing stateful servers, for example. When we modify the type of a record field at some point of the program, we might need to correct all update of that record field to reflect the type change. Therefore our second use case scenario tries to answer the following question: what are the expressions which update the value of a given record field?

An example query for the `loader_queue` field of the `state` record is as follows.

```
{ref | f <- files,
       r <- records f, name r == "state",
       fld <- fields r, name fld == "loader_queue",
       ref <- references fld, exprType (topExpr ref) == Record_update}
```

The same query in the Semantic Query Language, which serves as a standard in performance comparison, is the following.

```
files
  .records[name=state]
  .fields[name=loader_queue]
  .reference[.top_expr[type=record_update]]
```

5.3 Finding Suspicious Values

Due to the dynamic type system of Erlang, developers often need to investigate runtime errors related to the misuse of values. Once developers find errors such as `badmatch`, `function_clause`, which indicate failure in pattern matching, they might be interested where the given value originates from. A data-flow analysis is able to detect the flow of certain values, thus these are built into the query languages as well.

Our third use case is based on the following question: is there any expression with a given (incorrect) value that may reach a given function?

Let us consider the following example where we found the `previous_log` atom as a suspicious value that may cause a runtime error in the function `open_log/6`. We can write a query to find out whether the possible value (`origin`) of the first arguments is the atom `previous_log`:

```
{o | m <- modules, name m == "mnesia_log",
     f <- functions m, name f == "open_log", arity f == 6,
     p <- parameters f, paramIndex p == 1,
     o <- origin p, exprType o == Atom, exprValue o == "previous_log"}
```

The query in the Semantic Query Language is as follows.

```
mods[name=mnesia_log]
  .funs[name=open_log, arity=6]
  .params[index=1].origin[type=atom, value=previous_log]
```

If we have an editor with the file opened then we can also use a position-based query:

```
@expr.origin
```

These queries result two expressions in the mnesia database manager, both has the same value `previous_log`, which flows to the argument of `open_log/6`. Running a text-based search yields 14 expressions, and therefore it would take longer time to check all of them.

6 Comparison

In Section 2.3, we described our main concerns with the existing query language implementation and proposed a new methodology to build a maintainable implementation.

A quantitative comparison on the effect of the new implementation on the maintainability can be expressed in terms of person months used to implement certain features. The aforementioned variable binding feature implementation took 4.5-5 person months, and the result was still unsatisfying. In contrast, the design and implementation of the whole NequeLace from scratch took 2 person months.

It is worth noting that there is also a significant difference between the two implementations in terms of lines of code. The implementation of the RefactorErl language consists of 7772 lines, while the NequeLace implementation consists of 1980 lines (including comments and blank lines in both cases). Although the NequeLace implementation still misses several functions compared to the RefactorErl language, these functions contribute usually two or three lines.

6.1 Benchmark

Besides the general comparison on the development and the usability of the two languages, we also benchmarked performance of NequeLace implementation against that of RefactorErl Semantic Query Language to prove that NequeLace is usable in industrial-scale environment as well, like the RefactorErl language.

For this purpose, we loaded the Erlang distributed database mnesia, a code base consists of more than 24 thousand lines of code, into RefactorErl. We performed the benchmark on Debian GNU/Linux 9.2. The Haskell code was compiled with GHC 8.0.2, while RefactorErl[2] was compiled with Erlang OTP 19.2.

We used the queries presented in Section 5 to compare the execution times of the NequeLace implementation and the Semantic Query Language implementation. In Table 1 we summarized the results. An astute reader may recognize that NequeLace is slower. However, these differences are hardly perceptible. We conclude that NequeLace is suitable as a replacement for the query language of RefactorErl, while the development of the language is more straightforward, and the language is much more extensible.

Query	Execution time (s)		Deviation (s)	
	NequeLace	RefactorErl QL	NequeLace	RefactorErl QL
Dynamic call	0.046	0.017	0.0172	0.0007
Record update	0.083	0.005	0.0179	0.0001
Value origin	0.104	0.007	0.0204	0.0002

Table 1: Execution Time and Deviation

[2]RefactorErl is freely available to download at `http://plc.inf.elte.hu/erlang`

6.1.1 Slower Queries

Although the queries presented in Section 5 are fast and result a small amount of data, we chose them based on the most recent problems of our industrial partners. Thus, we present a long running query that operates on a larger data set as well. One example query collects the possible values of all expressions in a module. The result set contained 6644 different expressions. The NequeLace implementation needed 1 minute and 27 seconds on average.

```
{o | m <- modules, name m == "mnesia",
     f <- functions m,
     e <- expressions f,
     o <- origin e}
```

The RefactorErl implementation is faster, it took 1 minute 2 seconds on average to evaluate the query:

```
mods[name=mnesia].funs.expressions.origin
```

The main reason of the difference is the communication between Erlang and Haskell, which includes serialization and de-serialization. Although 25 seconds may seem as a big difference, it only makes NequeLace around 1.5 times slower. Compared to 1 minute of RefactorErl, an order of magnitude difference in execution time would indeed render the new language unusable. Fortunately, this is not the case.

7 Related Works

According to Philip Wadler [Jon87, chapter 7], the introduction of set builder notation in programming languages dates back to 1977, when Rod Burstall and John Darlington presented an early version of NPL, a functional programming language. Jacob T. Schwartz and others designed SETL [SDDS86], a language based on set theory, which includes set former expression, a variant of set builder notation.

The idea of using list comprehension as basis of a query language is not new. Philip Trinder[CT94, Tri92] argues that list comprehensions are suitable to express queries against relational, functional and object-oriented databases, and such queries are brief, clear and efficient.

List comprehension as query language made its way into modern languages. The .NET framework, for instance, includes Language Integrated Query (probably better known as LINQ). A query is a view of an input data sequence (e.g. a list of `Person` objects) that produces a new output sequence (a list of `String` objects, the names). When combined with object-relational mapping, an SQL database may also provide the input sequence.

Taking advantage of the meta-programming toolkit in Erlang (called parse transformation), a programmer can build database queries in form of a list comprehension. Such queries are called Query List Comprehensions (QLCs) [Heb13, p. 539] in Erlang terminology. For example, to query every user from a mnesia table `user` who lives in New York, one could write the following in an Erlang program.

```
Query = qlc:q([ U#user.name || U <- mnesia:table(user),
                               U#user.city == "New York" ]),
Usernames = qlc:eval(Query)
```

Database Supported Haskell (DSH) [UG15] is a domain-specific language embedded in Haskell. Programs in DSH describe data-parallel and data-intensive computations. They are written as list comprehensions and projections, which are translated into SQL queries.

Set builder notation is not the only way to traverse graphs. XPath and XQuery [Wal15], both are designed for XML document traversal, follow a different approach. XPath queries built on relation between nodes, called axis (e.g. child, descendant, parent) and predicates. XQuery is a superset of XPath which additionally includes functional language elements.

8 Conclusions

We have defined a good practice on query language design, and following that practice we have defined a simple, intuitive query language in purely functional style. We presented set builder expressions, which are universal in the sense that they serve as reusable template for query languages. The template can be applied in other contexts, the replacement set of built-in functions requires the most effort of all tasks.

There is plenty left to do. Pattern matching is popular feature among functional and logic languages. It can be utilized in the query language too. For instance, it would be much convenient if the programmer could search for tuples using pattern matching.

References

[Arm13] Joe Armstrong. *Programming Erlang.* The Pragmatic Bookshelf, 2nd edition, October 2013.

[BDN09] Ana Bove, Peter Dybjer, and Ulf Norell. A Brief Overview of Agda — A Functional Language with Dependent Types. In *Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 73–78. Springer-Verlag, 2009.

[BHH+11] István Bozó, Dániel Horpácsi, Zoltán Horváth, Róbert Kitlei, Judit Kőszegi, Máté Tejfel, and Melinda Tóth. RefactorErl, Source Code Analysis and Refactoring in Erlang. In *Proceeding of the 12th Symposium on Programming Languages and Software Tools*, Tallin, Estonia, 2011.

[CT94] Daniel K. C. Chan and Philip W. Trinder. Object comprehensions: A query notation for object-oriented databases. In David S. Bowers, editor, *Directions in Databases: 12th British National Conference on Databases*, volume 826 of *Lecture Notes in Computer Science*, pages 55–72. Springer Berlin Heidelberg, July 1994.

[Har16] Robert Harper. *Practical foundations for programming languages.* Cambridge University Press, 2016.

[Heb13] Fred Hebert. *Learn You Some Erlang for Great Good!* no starch press, January 2013.

[HHJW94] Cordelia Hall, Kevin Hammond, Simon Peyton Jones, and Philip Wadler. Type classes in Haskell. In *European Symposium On Programming*, volume 788 of *Lecture Notes in Computer Science*, pages 241 – 256. Springer Verlang, April 1994.

[Ive62] Kenneth E. Iverson. A programming language. In *Proceedings of Spring Joint Computer Conference*, AIEE-IRE, pages 345–351. ACM, 1962.

[Jon87] Simon Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice Hall, May 1987.

[Jon03] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report.* Cambridge University Press, April 2003.

[PS11] Manolis Papadakis and Konstantinos Sagonas. A PropEr integration of types and function specifications with property-based testing. In *Proceedings of the 2011 ACM SIGPLAN Erlang Workshop*, pages 39–50, New York, NY, September 2011. ACM Press.

[SDDS86] Jacob T. Schwartz, Robert B. K. Dewar, Edward Dubinsky, and Edith Schonberg. *Programming with Sets: An Introduction to SETL.* Springer-Verlag, October 1986.

[TBKH] Melinda Tóth, István Bozó, Judit Köszegi, and Zoltán Horváth. Static Analysis Based Support for Program Comprehension in Erlang. In Acta Electrotechnica et Informatica, Volume 11, Number 03, October 2011. Publisher: Versita, Warsaw, ISSN 1335-8243 (print), ISSN 1338-3957 (online), pages 3-10.

[Tri92] Philip W. Trinder. Comprehensions, a Query Notation for DBPLs. In *Proceedings of the Third International Workshop on Database Programming Languages*, DBPL3, pages 55–68. Morgan Kaufmann Publishers, 1992.

[UG15] Alexander Ulrich and Torsten Grust. The Flatter, the Better: Query Compilation Based on the Flattening Transformation. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD, pages 1421–1426. ACM, 2015.

[Wal15] Priscilla Walmsley. *XQuery.* O'Reilly Media, 2nd edition, December 2015.